5-5-2014

The At-Most-Once Problem: Definitions, Solutions and Impossibility Results

Sotirios Kentros
University of Connecticut - Storrs, sotirios.kentros@uconn.edu

The At-Most-Once Problem: Definitions, Solutions and Impossibility Results

Sotirios Kentros, PhD

University of Connecticut, 2014

Abstract

We introduce the at-most-once and strong at-most-once problems in the asynchronous shared memory model. The *at-most-once* problem requires that a set of n jobs is performed using m fail-prone processors, while maintaining at-most-once semantics. The *strong at-most-once* problem, imposes the additional restriction, that if no participating processors fail, all jobs must be performed. We rigorously define the problems and performance metrics, show upper bounds and impossibility results and provide deterministic and randomized solutions.

The at-most-once semantic is one of the standard safety guarantees for object access in decentralized systems. We investigate implementations of at-most-once access semantics in a model where a set of actions is to be performed by a set of failure-prone, asynchronous shared-memory processes. We formally introduce the *at-most-once* problem for performing a set of n jobs using m processors. We also introduce a notion of efficiency for at-most-once protocols, called *effectiveness*, used to classify algorithms. Effectiveness measures the number of jobs safely completed by an implementation. We prove an upper bound of $n - f$ on the effectiveness of any algorithm, where f the number of process crashes in the presence of an adversary.

We explore the feasibility of a strong effectiveness version of the at-most-once problem. The *strong at-most-once problem* is solved by an at-most-once algorithm when all tasks are performed, if no participating processes fail during the execution of the algorithm. We formally define the problem and prove that the strong at-most-once problem has consensus number 2, hence there exist no wait-free deterministic

solutions for the problem in asynchronous shared memory, using atomic read/write registers.

We prove that the upper bound on effectiveness of $n - f$ can be matched asymptotically in the two process setting. We then generalize our two-process solution in the multi-process setting with a hierarchical algorithm that achieves effectiveness of $n - \log m \cdot o(n)$. Moreover, we present and analyze a wait-free deterministic algorithm for the at-most-once problem, that provides for the first time nearly optimal effectiveness for the multi-process setting.

Finally we present the first randomized solution for the strong at-most-once problem. The solution is work optimal in expectation for a non-trivial number of participating processes. We also present an adaptive randomized solution for the Write-All problem. Our solution has high probability work that is linear plus some additive term that only depends on the number of participating processes k and not the size of the problem n or the total number of processes m.

The At-Most-Once Problem: Definitions, Solutions and Impossibility Results

SOTIRIOS KENTROS

Dipl.-Ing., Universiry of Patras, 2004

M.Sc., National Technical University of Athens, 2008

Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of Doctor of Philosophy

at the

University of Connecticut

2014

APPROVAL PAGE

Doctor of Philosophy Dissertation

The At-Most-Once Problem: Definitions, Solutions and Impossibility Results

Presented by

Sotirios Kentros

Major Advisor _____

Aggelos Kiayias

Associate Advisor _____

Alexander Russell

Associate Advisor _____

Alexander Shvartsman

Associate Advisor _____

Juan Garay

University of Connecticut

2014

Acknowledgments

I would like to express my gratitude to my advisor and friend Prof. Aggelos Kiayias for his continuous guidance, support and encouragement. Aggelos has been a perfect advisor, giving me confidence and independence to pursue my research interests. I have benefited greatly from his guidance and I would like to thank him for his active participation in developing me as a researcher and educator.

I would also like to thank Prof. Alexander Russell for working with me throughout my Ph.D. years. He has taken a keen interest in my research, always providing deep insight, through his extended understanding of the foundations of theoretical computer science. His multidimensional personality has been a constant inspiration to me. I have most enjoyed all classes I have taken with him, whether it were computer science courses or Aikido.

My sincere thanks to Prof. Alexander Shvartsman for introducing me to distributed computing. It was his and Prof. Kiayias' input that provided the motivation for this thesis. Prof. Shvartsman had a determining impact on the way I perceive research.

I am very grateful to Dr. Juan Garay for his support, encouragement and valuable insights in my research and dissertation.

I would like to thank Prof. Keith Barker. His contribution in my development as an educator has been invaluable, as it has been his continuous support and encouragement through my Ph.D. years.

I would also like to thank my collaborator and friend Prof. Chadi Kari. Through endless discussions and meetings he had a determining impact on my research and this dissertation.

I want to express my gratitude to my friends Iasonas Petras, Vasiliki Pappa, Ranita Ray, Solomon Berhe, Athanasios Bamis, Ioannis Kareklas and Bogdan Pasaniuc. Their

support, friendship and advise made my graduate school years a much better experience.

Finally I would like to thank my partner Dimitra, my parents Ilias and Anastasia and my sisters Despoina and Andromachi, for their support, understanding and motivation through the years.

My research has been supported in part by the State Scholarships Foundation of Greece.

Contents

1 Introduction, Motivation and Related Work

1.1 Introduction

The *at-most-once* semantic for object invocation is used to ensure that an operation accessing and altering the state of an object is performed no more than once. This semantic is among the standard semantics for remote procedure calls (RPC) and method invocations and it provides important means for reasoning about the safety of critical applications. Uniprocessor systems may trivially provide solutions for at-most-once semantics by implementing a central schedule for operations. The problem becomes very challenging for autonomous processes in a shared-memory system with concurrent invocations on multiple objects. Although at-most-once semantics have been thoroughly studied in the context of at-most-once message delivery [10, 34, 37, 46] and at-most-once process invocation for RPC [8, 35, 36, 37, 44], finding effective solutions for asynchronous shared-memory multiprocessors, in terms of how many at-most-once invocations can be performed by the cooperating processes, is largely an open problem.

This thesis brings attention to the at-most-once problem in multiprocessor settings. We believe that solving this problem using only atomic read/write memory, and without specialized hardware support, such as conditional writing, will provide a useful tool in reasoning about the safety properties of applications developed for a variety of multiprocessor systems, including those not supporting bus-interlocking instructions and multi-core systems. Specifically in the later years increased attention has been given toward *chip multiprocessing*, since clock speed has stopped being the way to increase performance. Because of the differences in each multi-core system, the asynchronous shared memory is becoming an important abstraction for arguing on the safety properties of parallel applications in such systems. In the next years we expect chip multiprocessing to

appear in a wide range of applications, many of which will have components that need to satisfy at-most-once semantics in order to guarantee safety. Such applications may include autonomous robotic devices, robotic devices for assisted living, automation in production lines or medical facilities. In such applications performing specific tasks at-most-once may be of paramount importance for safety of patients, or the workers in a facility, or the devices themselves. Such tasks could be the triggering of a motor in a robotic arm, the activation of the X-ray gun in an X-ray machine, or supplying a dosage of medicine to a patient. In order to further illustrate the importance of the at-most-once semantic, we point out that in *write-all* [25] a dual to the at-most-once problem, there exist specific tasks that in an execution could be executed as many times as the total number of processors in the system.

We explore at-most-once implementations for asynchronous shared-memory processors that are prone to crashes. We model accesses to objects as jobs, where the correctness demands that each job is performed at-most-once. Any processor is able to perform any job and we aim to maximize the total number of performed jobs while preserving the at-most-once semantics. We define the notion of *effectiveness* used to assess the efficiency of solutions for the problem. Effectiveness measures the number of jobs performed using at-most-once semantics as a function of the number of jobs, the number of processors, and the number of crashes. We provide tight upper bounds for effectiveness, and we introduce various deterministic, wait-free algorithms that solve the problem. The first two are formulated for two processors. The third algorithm is stated for an arbitrary number of processors and it uses a two-processor solution as a building block. We present rigorous analyses of the algorithms' work, space complexity, and effectiveness and we prove that the multi-processor algorithm has effectiveness $(n^{\frac{1}{\log m}} - 1)^{\log m}$ and work complexity $O(n + m \log m)$, where n is the number of jobs and m the number of processes [29]. Work

complexity counts the total number of basic operations performed by the processes.

Because the multi-processor solution presented in [29] does not scale well in terms of effectiveness, as the number of processors increases, we explore a different algorithmic strategy. This gives us a deterministic, wait-free solution for the at-most-once problem that has near optimal effectiveness. Specifically the effectiveness of the solution is $n-(2m-2)$, which comes close to an additive term of m to the upper bound over all possible algorithms [27, 28]. This second solution scales much better in terms of the number of processes. Moreover we demonstrate how to construct an algorithm that has effectiveness $n-\mathrm{O}(m^2 \log n \log m)$ and work complexity $\mathrm{O}(n+m^{3+\varepsilon} \log n)$, and is both effectiveness and work optimal when $m=\mathrm{O}(\sqrt[3+\varepsilon]{n/\log n})$, for any constant $\varepsilon > 0$. This algorithm has both asymptotically optimal effectiveness and optimal work for a non-trivial number of processes. Finally we show how to use this algorithm in order to solve the *Write-All* problem [25] with optimal work complexity for a non-trivial number of processes.

Motivated by the difficulty of implementing wait-free deterministic solutions that are effectiveness optimal, we introduce the *strong at-most-once problem* and study its feasibility in [26]. The strong at-most-once problem refers to the setting where effectiveness is measured only in terms of the jobs that need to be executed and the processes that took part in the computation and crashed. The strong at-most-once problem demands solutions that are adaptive, in the sense that the effectiveness depends only on the behavior of processes that participate in the execution. In this manner trivial solutions are excluded and, as we demonstrate herein, processes have to solve an agreement primitive in order to make progress and provide a solution for the problem. In the present work we prove that the strong at-most-once problem has consensus number 2 as defined by Herlihy [23]. As a result, there exists no wait-free deterministic solution for the strong at-most-once problem in the asynchronous share memory model, using atomic read/write registers. This

explains, via impossibility, the lack of effectiveness optimal deterministic solutions for the at-most-once problem.

Then, we present [26] a randomized k-adaptive solution for the strong at-most-once problem that has optimal strong effectiveness of $n - f_k$ and expected work complexity of $O(n + k^{2+\varepsilon} \log n)$ for any small constant ε. Sometimes k is called the contention of an execution and denotes the number of processes that take part in an execution of the algorithm, f_k denotes the number of the k processes that crash in an execution. Our solution is the first fully adaptive randomized solution (both in terms of effectiveness and expected work complexity) for the strong at-most-once problem. Expected Work complexity counts the expected total number of basic operations performed by the processes. Moreover our solution is anonymous, in that it does not rely on the names of processes.

Finally we present a k-adaptive randomized solution for the Write-All problem that has work complexity of $O(n + k^2 log^3 k)$ with high probability. This algorithm is the first solution for the Write-All problem that has linear work plus some additive term that only depends on the number of participating processes k and not the size of the problem n or the number of processes m that can participate in the algorithm. We further demonstrate how to construct a solution for the strong at-most-once problem, using this algorithm, that has work complexity $O(n + k^2 log^3 k)$ with high probability.

1.2 Related Work

A wide range of works study at-most-once semantics in a variety of settings. At-most-once message delivery [10, 34, 37, 46] and at-most-once semantics for RPC [8, 35, 36, 37, 44], are two areas that have attracted a lot of attention. Here the problem studied is different from the one we consider. Both in at-most-once message delivery and RPCs, we have two entities (sender/client and receiver/server) that communicate by message

passing. Any entity may fail and recover and messages may be delayed or lost. In the first case one wants to guarantee that duplicate messages will not be accepted by the receiver, while in the case of RPCs, one wants to guarantee that the procedure called in the remote server will be invoked at-most-once [45].

Di Crescenzo and Kiayias in [12] (and later Fitzi *et al.* [14]) demonstrate the use of the semantic in message passing systems for the purpose of secure communication. Driven by the fundamental security requirements of *one-time pad* encryption, the authors partition a common random pad among multiple communicating parties. Perfect security can be achieved only if every piece of the pad is used at most once. The authors show how the parties maintain security while maximizing efficiency by applying at-most-once semantics on pad expenditure.

One can also relate the at-most-once problem to the consensus problem [13, 23, 40, 33]. Indeed, consensus can be viewed as an at-most-once distributed decision. Another related problem is process renaming, see Attiya *et al.* [6] where each process identifier should be assigned to at most one process. Proving that the strong at-most-once problem has consensus number 2 further demonstrates this relationship.

The at-most-once problem is in some sense a dual problem to the Write-All problem for the shared memory model [4, 9, 11, 21, 25, 32, 41]. First presented by Kanellakis and Shvartsman [25], the Write-All problem is concerned with performing each job *at-least-once*. Most of the solutions for the Write-All problem in the asynchronous share memory model, exhibit super-linear work even when $m \ll n$. Malewicz [41] was the first to present an asynchronous solution for the Write-All problem that has linear work for a non-trivial number of processors. The algorithm presented by Malewicz [41] has work $O(n + m^4 \log n)$ and uses test-and-set operations. Later Kowalski and Shvartsman [32] presented an asynchronous solution for the Write-All problem that for any constant ε

has work $O(n + m^{2+\varepsilon})$. Their algorithm uses a collection of q permutations with contention $O(q \log q)$ for a properly chosen constant q and does not rely on test-and-set operations. Although an efficient polynomial time construction of permutations with contention $O(q \operatorname{polylog} q)$ has been developed by Kowalski *et al.* [30], it is not known to date how to construct permutations with contention $O(q \log q)$ in polynomial time. Recently Alistarh *et al.* [2] show that there exists a deterministic asynchronous algorithm for the Write-All problem with work $O(n + m \log^5 n \log^2 \max(n,m))$, by derandomizing their randomized solution for the problem. Their solution is a breakthrough in terms of bridging the gap between the $\Omega(n + m \log m)$ lower bound for the Write-All problem and known deterministic solutions, but is so far existential. For a detailed overview of research on the Write-All problem, we refer the reader to the books by Georgiou and Shvartsman [18, 19]. In [27, 28] we demonstrate how our iterative At-Most-Once solution can be converted into an asynchronous solution for the Write-All problem with work complexity $O(n + m^{3+\varepsilon} \log n)$ for any constant $\varepsilon > 0$.

With respect to asynchronous randomized solutions, Martel and Subramonian [42] present a randomized solution for the Write-All problem that does optimal $O(n)$ work when the number of processes is less than $\frac{n}{\log n}$. Their solution assumes an oblivious adversary, which is a weaker adversary than the strong adaptive adversary we use in this dissertation when examining randomized algorithms. When it comes to a strong adaptive adversary, Anderson and Woll [4] provide a $O(n \log m)$ solution for $n = m^2$ write-all cells and m processes. As mentioned above, recently Alistarh *et al.* [2] provide a randomized algorithm for a strong adaptive adversary with work $O(n + m \log m \log^2 n)$, which is the best known result in terms of randomized algorithms. It is easy to see that their solution would perform $O(n + k \log k \log^2 n)$ work with high probability if examined on an adaptive setting where k is the number of participating processes. The last result presented in this

dissertation, provides the first solution for the Write-All problem that has linear work plus some additive term that only depends on the number of participating processes k and not the size of the problem n or the number of processes m that can participate in the algorithm. In that respect it improves over the known adaptive solutions for the Write-All problem.

Finally, following our work [29], Hillel [24] provided a probabilistic algorithm in the same setting with optimal effectiveness and expected work complexity $O(nm^2 \log m)$ by employing a probabilistic multi-valued consensus protocol as a building block. With proper analysis it can be shown that the probabilistic solution of Hillel [24], is a solution for the strong at-most-once problem.

We note that the at-most-once problem becomes simpler when shared-memory is supplemented by some type of read-modify-write operations. For example, one can associate a *test-and-set* bit with each task, ensuring that the task is assigned to the only process that successfully sets the shared bit. An effectiveness optimal implementation can then be obtained from any Write-All solution. This is intuitive since the strong at-most-once problem has consensus number 2, the same as test-and-set. In Buss *et al.* [9] they demonstrate a lower bound of $n + \Omega(m \log n)$ for $3 \leq m \leq n$ in the asynchronous shared memory setting, when atomic primitives such as compare-and-swap or test-and-set are used to access shared memory. This implies a lower bound on the work complexity of Strong At-Most-Once asynchronous solutions in the presence of test-and-set operations. In this thesis, when devising deterministic solutions, we deal with the more challenging setting where algorithms use atomic read/write registers. For asynchronous randomized solutions for the Strong At-Most-Once problem will use randomized solutions for test-and-set, as a building block.

1.3 Contributions

The goal of this dissertation, is to explore the feasibility and efficiency of solutions that satisfy the at-most-once semantic in the shared-memory model with asynchronous processors prone to crash failures. The core of this dissertation is based on the results presented in [28, 26, 27, 29]. The at-most-once problem is formulated for m processors and n jobs, where any processor can perform any job, provided that no job is performed more-than-once. Note that in such a setting it is impossible to distinguish between a slow and a crashed processor. Consequently it is impossible to determine whether a processor delays while performing a job or if it crashed before performing the job. This means that generally some jobs may never be performed. For the strong at-most-once problem, we consider that only k of the m processors participate in the execution from which f_k may later crash. The number of participating processes k is not known to the algorithm. Our contributions are as follows.

1. We define the *at-most-once* problem and the correctness properties to be satisfied by any solution. We introduce a complexity measure we call *effectiveness*. This measure describes the number of jobs completed (at-most-once) by an implementation, as a function of the overall number of jobs n, the number of processors m, and the number of processor crashes f [29].

2. We present an upper bound for the effectiveness of any at-most-once implementation. In particular, we prove that no at-most-once solution may achieve effectiveness better than $n - f$ [29].

3. We define the *strong at-most-once problem*, as the problem of solving the at-most-once problem with effectiveness that is a function of the jobs that need to be executed n and the processes that took part in the computation (took a least one step

in the computation) and crashed f_k [26]. From the upper bound on effectiveness for all algorithms (see [29]), we get that an effectiveness of $n - f_k$ for the strong at-most-once problem is optimal. The importance of the strong at-most-once problem is that it does not allow for solutions, where some jobs are preassigned to specific processes.

4. We show [26] that the strong at-most-once problem has consensus number 2 (see [23]), and thus there exists no wait-free deterministic solution for the strong at-most-once problem using read/write atomic registers. Moreover we observe that the strong at-most-once problem belongs in the *Common*2 class as defined by Afek *et al.* [1].

5. We provide two algorithms that solve the at-most-once problem for 2 processors. The algorithms use a collision-avoidance approach. The importance of these algorithms is twofold: a) they can be used as building blocks to construct general implementations for larger number of processors, and b) they achieve optimal effectiveness. The algorithms differ substantially in their space requirements and work complexity, demonstrating the trade-offs between efficiency and space. We analyze work, space, and effectiveness. [29]

6. We present a multi-processor algorithm, that employs one of our two-processor algorithms as a building block. We prove the correctness of the algorithm, and we perform rigorous analysis of its effectiveness of $n - \log m \cdot o(n)$, and its work and space complexity. [29]

7. We present a multi-processor algorithm, that uses a different strategy. The algorithm is parametrized by $\beta \geq m$ and has effectiveness $n - \beta - m + 2$. If $\beta < m$ the correctness of the algorithm is still guaranteed, but the termination of the algorithm

cannot be guaranteed. For $\beta = m$ the algorithm has effectiveness of $n - 2m + 2$, which is optimal up to an additive term of m. We further prove that for $\beta \geq 3m^2$ the algorithm has work complexity $O(nm \log n \log m)$ [27, 28].

8. We show how to use the previous algorithm with $\beta = 3m^2$, in order to construct an iterative algorithm, which for any constant $\varepsilon > 0$, has effectiveness of $n - O(m^2 \log n \log m)$ and work complexity $O(n + m^{3+\varepsilon} \log n)$. This is both asymptotically effectiveness-optimal and work-optimal for any $m = O(\sqrt[3+\varepsilon]{n/\log n})$ [27, 28].

9. We demonstrate [27, 28] how to use the iterative algorithm mentioned above in order to solve the Write-All problem with work complexity $O(n + m^{3+\varepsilon} \log n)$ for any constant $\varepsilon > 0$. Our solution improves on the algorithm of Malewicz [41], which solves the Write-All problem for a non-trivial number of processes with optimal (linear) work complexity, in two ways. First our solution is work optimal for a wider range of m, namely for any $m = O(\sqrt[3+\varepsilon]{n/\log n})$ compared to the $m = O(\sqrt[4]{n/\log n})$ of Malewicz . Second our solution does not assume the test-and-set primitive used by Malewicz [41], and relies *only* on atomic read/write memory. There is also a Write-All algorithm due to Kowalski and Shvartsman [32], which is work optimal for a wider range of processors m than our algorithm, specifically for $m = O(\sqrt[2+\varepsilon]{n})$. However, their algorithm uses a collection of q permutations with contention $O(q \log q)$, while it is not known to date how to construct such permutations in polynomial time. The solution from Alistarh *et al.* [2] show that there exists a deterministic algorithm for the Write-All problem with work $O(n + m \log^5 n \log^2 \max(n, m))$, by derandomizing their randomized solution for the problem. This solutions is also work optimal for a wider range of processors m than our algorithm, but their solution is existential, while ours explicit.

10. We present and analyze a randomized algorithm that solves the strong at-most-once problem [26]. The algorithm is anonymous, wait-free and k-adaptive, in the sense that both effectiveness and work complexity depend on k, the number of processes that participate in the execution and not m the total number of processes. The algorithm uses randomized test-and-set as a building block, both for guarantying the at-most-once semantic and for bookkeeping. In terms of bookkeeping, it is used in order to detect collisions between processes and facilitate the transfer of knowledge on which jobs have already been performed. Our solution uses the *RatRace* algorithm from Alistarh *et. al.* [3] for the randomized test-and-set operations. We show that the algorithm has strong optimal effectiveness and expected work complexity $O(n+k^{2+\varepsilon}\log n)$ for any small constant ε. The algorithm improves over the solution of Hillel [24] which can be shown to have strong optimal effectiveness and expected work complexity $O(nm^2\log m)$. We also demonstrate how to modify the randomized algorithm in order to solve the Write-All problem. To our knowledge our algorithm is the first k-adaptive randomized solution for the Write-All problem. Following our solution Alistarh *et al.* [2] device a randomized algorithm for the Write-All with work complexity $O(n+m\log m\log^2 n)$. With some modifications this algorithm can be used to solve the strong at-most-once problem. We conjecture that its work can be bounded by $O(n+k\log^2 k\log^2 n)$ with high probability. Which is an improvement over our solution.

11. We present and analyze a randomized k-adaptive Write-All algorithm that has work $O(n+k^2\log^3 k)$ with high probability. To the best of our knowledge, this result provides the first solution for the Write-All problem that has linear work plus some additive term that only depends on the number of participating processes k and not the size of the problem n or the number of processes m that can participate in

the algorithm. To be more specific, in the solution by Alistarh *et al.* [2] there is a $\log^2 n$ factor in the $k \log^2 k \log^2 n$ additive term. In the deterministic randomized algorithms by Malewicz [41], Kowalski and Shvartsman [32] and Kentros and Kiayias [27, 28] the additive terms have polynomial factors of the number of processes m that can participate in the algorithm.

The algorithms in [29] are motivated by the *Write-All* algorithms from [9, 21], while the algorithm in [27, 28] is motivated by a *renaming* algorithm from [6], although the problem itself and the correctness criteria are quite different. Our work can be viewed as complementary to [12] that considers a similar problem in message-passing models. Here we use a shared-memory model instead.

1.4 Structure

This dissertation is structured as follows: Section 2 presents the asynchronous shared memory model we are using in order to define the at-most-once and strong at-most-once problem, to provide and analyze solutions. Moreover the at-most-once problem, the measure of effectiveness and the strong at-most-once problem are defined in the same Section. Section 3 presents the upper bound on all possible algorithms for the effectiveness of at-most-once solutions (from Kentros *et al.* [29]). Moreover in the same Section impossibility results on wait-free deterministic solutions for the strong at-most-once problem, are presented (from Kentros *et al.* [26]). In Section 4 we present deterministic collision avoidance based solutions for the at-most-once problem (from Kentros *et al.* [29]). We start by two 2-process solutions which we then use as building blocks for a multiprocess solution. In Section 5 a near optimal wait-free deterministic solutions called algorithm KK_β is presented and analyzed(from Kentros and Kiayias [27, 28]). Subsequently in Section 6 two iterative algorithms base on KK_β from Section 5 are presented (the algorithms

appeared in Kentros and Kiayias [27, 28]). The first solves the at-most-once problem, while the second provides a solutions for the write-all problem. We continue with Section 7 which describes a randomized wait-free k-adaptive algorithm named RA for the strong at-most-once problem (from Kentros *et al.* [26]). Then in Section 8 a randomized k-adaptive Write-All algorithm named ARTA is presented. In Section 9 we propose two specific directions, in order to expand the results presented in the previous sections. Finally we conclude in Section 10 with open problems arising from the work in this doctoral dissertation.

2 Model, Definitions and Measures of Efficiency

We define our model, the at-most-once problem, the strong at-most-once problem and measures of efficiency.

2.1 Model and Adversary

We model a multi-processor as m asynchronous, crash-prone processes. For deterministic solutions, processes have unique identifiers from some set $\mathcal{P}$. Shared memory is modeled as a collection of atomic read/write memory cells, where the number of bits in each cell is explicitly defined. In the algorithm presented in Section 8, the shared memory additionally supports atomic test-and-set objects.

A test-and-set object atomically sets a memory location to the value 1 (or set) and returns 1 (or success) if the original value of the memory location was 0, or 0 (fail) if the original value of the memory location was 1. We say that a process wins or succeeds in a test-and-set if it returns 1; otherwise, the process loses the test-and-set. A test-and-set is acquired if it has been won by a process. All memory locations associated with

test-and-set operations are initialized to 0.

We use the *Input/Output Automata* formalism [40, 39] to specify and reason about algorithms in Sections 4 and 5; specifically, we use the *asynchronous shared memory automaton* formalization [40, 20]. The algorithms in Sections 6, 7 and 8, are presented using pseudo-code.

For completeness we describe the *Input/Output Automata* formalism we use. Each process p is defined in terms of its states $states_p$ and its actions $acts_p$, where each action is of the type *input*, *output*, or *internal*. We further distinguish between two different kinds of internal actions: (i) those where a process interacts with the shared memory called *shared memory actions* and (ii) those that no interaction with the shared memory takes place. The *shared memory actions* are further distinguished in *Read* or *Write* actions. Read and Write internal actions will be clearly denoted in the signature of an automaton. A subset $start_p \subseteq states_p$ contains all the start states of p. Each shared variable x takes values from a set V_x, among which there is $init_x$, the initial value of x.

We model an algorithm A as a composition of the automata for each process p. Automaton A consists of a set of states $states(A)$, where each state s contains a state $s_p \in states_p$ for each p, and a value $v \in V_x$ for each shared variable x. Start states $start(A)$ is a subset of $states(A)$, where each state contains a $start_p$ for each p and an $init_x$ for each x. The actions of A, $acts(A)$ consists of actions $\pi \in acts_p$ for each process p. A transition is the modification of the state as a result of an action and is represented by a triple (s, π, s'), where $s, s' \in states(A)$ and $\pi \in acts(A)$. The set of all transitions is denoted by $trans(A)$. Each action in $acts(A)$ is performed by a process p, thus for any transition (s, π, s'), s and s' may differ only with respect to the state s_p of process p that invoked π and potentially the value of the shared variable that p interacts with during π (if π is a Write action). We also use triples $(\{vars_s\}, \pi, \{vars_{s'}\})$, where $vars_s$ and $vars_{s'}$ are sub-

sets of variables in s and s' respectively, as a shorthand to describe transitions without having to specify s and s' completely; here $vars_s$ and $vars_{s'}$ contain only the variables whose value changes as the result of π, plus possibly some other variables of interest.

We say that states s and t in $states(A)$ are *indistinguishable* to process p if: 1) $s_p = t_p$, and 2) the values of all shared variables are the same in s and t. Now, if states s and t are indistinguishable to p and $(s,\pi,s') \in trans(A)$ for $\pi \in acts_p$, then $(t,\pi,t') \in trans(A)$, and s' and t' are also indistinguishable to p.

An *execution* fragment of A is either a finite sequence, $s_0,\pi_1,s_1,\ldots,\pi_r,s_r$, or an infinite sequence, $s_0,\pi_1,s_1,\ldots,\pi_r,s_r,\ldots$, of alternating states and actions, where $(s_k,\pi_{k+1},s_{k+1}) \in trans(A)$ for any $k \geq 0$. If $s_0 \in start(A)$, then the sequence is called an *execution*. The set of executions of A is *execs(A)*. We say that execution α is *fair*, if α is finite and its last state is a state of A where no locally controlled action is enabled, or α is infinite and every locally controlled action $\pi \in acts(A)$ is performed infinitely many times or there are infinitely many states in α where π is disabled. The set of fair executions of A is $fairexecs(A)$. An execution fragment α' *extends* a finite execution fragment α of A, if α' begins with the last state of α. We let $\alpha \cdot \alpha'$ stand for the execution fragment resulting from concatenating α and α' and removing the (duplicated) first state of α'.

For two states s and s' of an execution fragment α, we say that state s *precedes* state s' and we write $s < s'$ if s appears before s' in α. Moreover we write $s \leq s'$ if state s either precedes state s' in α or the states s and s' are the same state of α. We use the term precedes and the symbols $<$ and $\leq$ in a same way for the actions of an execution fragment. We use the term precedes and the symbol $<$ if an action π appears before a state s in an execution fragment α or if a state s appears before an action π in α. Finally for a set of states S of an execution fragment α, we define as $s_{max} = \max S$ the state $s_{max} \in S$, s.t. $\forall s \in S$, $s \leq s_{max}$ in α.

We model process crashes by action stop_p in $acts(A)$ for each process p. If stop_p appears in an execution α then no actions $\pi \in acts_p$ appear in α thereafter. We then say that process p *crashed*.

We consider an *omniscient* or *on-line adversary*: in the sense that the adversary has complete knowledge of the computation it is affecting, and it makes instant dynamic decisions on how to affect the computation. The adversary controls asynchrony and crashes. This is modeled by allowing the adversary to make all scheduling decisions, essentially arranging the order processes take actions. The adversary can base its next scheduling decision, on the execution so far, that is on the past and current values of the shared memory, and the past and current local states of all the processes.

In randomized algorithms (Sections 7 and 8) processes have also access to local random coin-flips. The adversary we consider in these Sections, has also access to the results of local coin-flips so far. Notice, however, that the adversary does not know the results of local coins that were not yet flipped (future coin flips). This adversary is called an *adaptive* adversary.

We would like to point out here that the *omniscient* adversary for the deterministic algorithms and the *adaptive* adversary for randomized algorithms are equivalent. In randomized algorithms the past results of local coin-flips by a process p are reflected in the state of a process p. So the fact that the adversary bases its next scheduling decision on the execution so far, having knowledge of the past and current local states of all the processes, implies that the adversary has knowledge of the past results of local coin-flips. For this reason in the definitions that follow in Section 2.2 we do not specifically define the adversary used.

In Sections 7 and 8 we are interested in the number of participating processes k. For Sections 7 and 8, we distinguish the crashes the adversary can cause, in $m-k$ $(k > 1)$

crashes at the beginning of the execution (before processes can perform any action) and $f_k < k$ crashes during the execution after a process performs at least one action.

In all algorithms we examine, the total number of crashes allowed is $f < m$. We denote by $fairexecs_f(A)$ all fair executions of A with f crashes. Also we denote by $fairexecs_{f,f_k}(A)$, with $f \geq f_k$, all fair executions of A where $k > f_k$ processes take at least one step in the execution and exactly f_k of the k processes crash. Clearly $f = n - k + f_k$.

2.2 At-Most-Once Problem, Strong At-Most-Once Problem and Metrics

We consider algorithms that perform a set of *tasks* or *jobs* (we will use both terms interchangeably). Let A be an algorithm specified for m processes from set $\mathcal{P}$, and for n jobs with unique ids from set $\mathcal{J} = [1 \ldots n]$. We assume that there are at least as many jobs as there are processes, i.e., $n \geq m$. We model the performance of job j by process p by means of action $\mathsf{do}_{p,j}$. For a sequence c, we let $len(c)$ denote its length, and we let $c|_\pi$ denote the sequence of elements π occurring in c. Then for an execution α, $len\left(\alpha|_{\mathsf{do}_{p,j}}\right)$ is the number of times process p performs job j. Finally we denote by $F_\alpha = \{p | stop_p \text{ occurs in } \alpha\}$ the set of crashed processes in execution α. We now define the number of jobs performed in an execution, the *at-most-once problem* and *effectiveness*.

Definition 2.1. *For execution α let $\mathcal{J}_\alpha = \{j \in \mathcal{J} | \mathsf{do}_{p,j}$ occurs in α for some $p \in \mathcal{P}\}$. The total number of jobs performed in α is defined to be $Do(\alpha) = |\mathcal{J}_\alpha|$.*

Definition 2.2. *Algorithm A solves the at-most-once problem if for each execution α of A we have $\forall j \in \mathcal{J} : \sum_{p \in \mathcal{P}} len\left(\alpha|_{\mathsf{do}_{p,j}}\right) \leq 1$. We call any such execution α an at-most-once execution.*

We next define *effectiveness*, that counts the number of jobs performed by an algo-

rithm in the worst case.

Definition 2.3. *$E_A(n,m,f) = \min_{\alpha \in fairexecs_f(A)}(Do(\alpha))$ is the **effectiveness** of algorithm A, where m is the number of processes, n is the number of jobs, and f is the number of crashes. An alternate definition is $E_A(n,m,f,f_k) = \min_{\alpha \in fairexecs_{f,f_k}(A)}(Do(\alpha))$, where $f_k \leq f$ the number of processes that crashed in an execution after taking at least one step.*

A trivial algorithm can solve the at-most-once problem by splitting the n jobs in groups of size $\frac{n}{m}$ and assigning one group to each process. Such a solution has effectiveness $E(n,m,f) = (m-f) \cdot \frac{n}{m}$ (consider an execution where f processes fail at the beginning of the execution).

We also define the *strong at-most-once problem*. The strong at-most-once problem, requires that only processes that participate in an execution and fail can block an at-most-once job. Trivial solutions for the at-most-once problem, such as the one described above are not valid solutions for the strong at-most-once problem. We show that the strong at-most-once problem has consensus number 2 as defined in [23].

Definition 2.4. *Algorithm A solves the **strong at-most-once problem** if algorithm A solves the at-most-once problem and there exists function $\varphi()$, such that $\varphi(0) = 0$ and for all f, f_k, with $m > f \geq f_k$, $E_A(n,m,f,f_k) = n - \varphi(f_k)$.*

The difference between the at-most-once problem and the strong at-most-once problem is that the latter requires that algorithms are implemented, such that in all initial states of the algorithm, no job is preassigned in a process. In other words, no process can start by performing a job, without first getting information about the current state of the execution. Moreover any job, may be performed by any process in some execution of the algorithm. In that sense, the 2-process effectiveness optimal algorithm presented in Section 4, is not a solution for the strong at-most-once problem, since the job with id 1 (resp. with id n) cannot be performed by the process with pid 1 (resp. pid 0).

Work complexity measures the total number of basic operations (comparisons, additions, multiplications, shared memory reads and writes) performed by an algorithm. We assume that each internal or shared memory cell has size $O(\log n)$ bits and performing operations involving a constant number of memory cell costs $O(1)$ (in some cases we may consider that such operations may have cost linear to the number of bits of particular memory cells, when this is the case, it will be explicitly stated). This is consistent with the way work complexity is measured in previous related work [25, 32, 41].

Definition 2.5. *The **work** of algorithm A, denoted by W_A, is the worst case total number of basic operations performed by all the processes of algorithm A.*

For randomized algorithms, expected work complexity measures the expected total number of basic operations performed by an algorithm. High probability work complexity measures the total number of basic operations with probability $1 - k^{-a}$ for any constant a. We are interested in adaptive randomized algorithms and thus we want the expected (or high probability) work complexity to be expressed as a function of n the total number of jobs and k the number of processes that participate in the the algorithm.

Definition 2.6. *The **expected work** of algorithm A, denoted by W_A, is the expected total number of basic operations performed by all the processes of algorithm A.*

Definition 2.7. *For any constant a the **high probability work** of algorithm A, denoted by W_A, is the total number of basic operations performed by all the processes of algorithm A with probability greater than $1 - k^{-a}$.*

Note that we overload W_A for all different types of work complexity (worst case, expected and high probability). During the analysis of algorithms we will explicitly state which work complexity we are using.

Space complexity measures the memory space used by the algorithm. In some cases we may consider only the shared memory used by the algorithm, or both the shared and local memory used. Also we may count the number of bits used in memory, or the number of memory cells, where each memory cell has $O(\log n)$ bits. If space complexity is analyzed, we will state explicitly which of the above conventions we use.

We will prove that the strong at-most-once problem has consensus number 2. Informally a *consensus protocol* is a system of n processes that communicate though a set of shared objects. Each process start with an input value. Processes communicate with one another by applying operations to the shared objects and eventually agree on a common input value and halt. A consensus protocol is required to be a) consistent: distinct processes never decide on distinct values, b) wait-free: each process decides after a finite number of steps, c) valid: the common decisions value is the input to some process (from Herlihy [23]). We say that an object X *solves n-process consensus*, if there exist a consensus protocol for n-process that uses a set of objects X and read/write registers, where the X can be initialized in any state. We provide the definition of consensus number form Herlihy [23].

Definition 2.8. *The consensus number for X is the largest n for which X solves n-process consensus. If no largest n exists, the consensus number is said to be infinite.*

We further use the following definition for the Write-All problem in Section 8.

Definition 2.9. *An algorithm solves the Write-All problem for n tasks with k participating processes, if the following three conditions hold:*

a) Termination - *Each participating process terminates after a finite number steps,*

b) Validity - *When the first process p terminates in an execution α, for all tasks $i \in \mathcal{J}$, there exists process q and action $\mathsf{do}_{q,j}$ that preceded in α the state process p terminated,*

c) Certification - *When any q process terminates in an execution α, it knows that all tasks have been performed.*

We will also need the following definition:

Definition 2.10. *Let S be a set of elements with unique identifiers. We define as the rank of element $x \in S$ and we write $[x]_S$, the rank of x if we sort in ascending order the elements of S according to their identifiers.*

2.3 Open Problems

There are quite a few topics left open concerning both modeling and definitions. The asynchronous shared memory model is a high level abstraction that covers most multi-processor and multi-core architectures. Still there are many architecture specific models that are general enough to have wide applicability and but different enough to benefit from different algorithmic approaches. Architecture aware approaches can lead in more efficient designs both in terms of effectiveness and work. Exploring the (strong) at-most-once problem in *hierarchical shared memory* may lead in surprising new results. Of particular interest should be *Non-Uniform Memory Access* multiprocessors. Moreover one can examine how *cache coherency* protocols could affect the design of algorithms. Different *consistency models* for cache coherency could benefit from different algorithmic strategies.

Another direction lies in examining simpler shared memory models. Examining the problem in synchronous shared memory with fail-stop crashes, could provide good useful intuition on the fundamental difficulties of the at-most-once problem.

A questions lies on how failure detection oracles can influence the design of solutions for the at-most-once problem. The upper bound on effectiveness for the at-most-once problem (Corollary 3.3) will still hold, but would it become easier to device tight solutions

in terms of optimal effectiveness?

In terms of modeling an obvious direction is towards message passing systems. There is a wide range of challenges in such an endeavor. New definitions are needed for the problem, as well as examination of what will be a meaningful message passing model for the problem. How fragmentation can affect solutions and whether it can be circumvented or not. One way to approach this problem will be to emulate and evaluate shared memory algorithms in the message passing system, using similar techniques with the ones used in [31]. Working on the message passing model will be a continuation and extension of the work in [12]. Finally, since in order to simplify programming, message passing systems resort in using middle-ware architectures that implement asynchronous shared memory, it is interesting to explore asynchronous shared memory models that support weaker primitives than the atomic read/write registers we have used so far.

In regard to the problem definition, there are few things to examine. The natural generalization of the at-most-once problem is the at-most-k notion, where k is either a constant, or depends on m the number of processors. One then needs to examine how the impossibility results and the lower bounds for the at-most-once problem transfer in the new setting. A strong notion of the at-most-k would also be worth examining. A different direction would be the definition of a weak at-most-once, or a weak at-most-k problem where the safety semantics can be violating to some controlled extend. Such a problem may have applicability in real systems, where jobs are at-most-once in nature, but one can afford some violation of the semantic.

It is also interesting to explore under what model a solution to a *do-all and at-most-k* problem can be achieved and how strong such a primitive is in terms of its consensus number. From the impossibility results on the at-most-once such a problem is clearly not solvable if k failures are allowed. So a question arises on whether there exist models,

where this impossibility can be lifted.

Finally dynamic versions of the at-most-once problem can be examined, where either the tasks are dispatched dynamically, or the processes arrive in a dynamic way, or both. Such a setting imposes different challenges than the static model and is closer to real applications.

3 Impossibility Results

3.1 Upper Bound

We show that any algorithm that solves the at-most-once problem in the presence of up to f crashes has effectiveness $E \leq n - f$. While the proof is subtle, the result itself is intuitive based on the observation that one cannot distinguish a crashed process from a slow one. If an algorithm assigns job j to process p, and the process crashes, the algorithm is unable to revoke the job and assign it to another process, since process p may simply be slow and it may ultimately perform job j, violating at-most-once semantics

Recall that in our setting we have at least as many jobs as processes ($n \geq m > f$). For our proofs we consider only algorithms that satisfy Condition 1 below requiring that the algorithm is able to perform at least one job. Also let us denote by $F_\alpha = \{p \mid \mathsf{stop}_p \text{ occurs in } \alpha\}$ the set of crashed processes in execution α.

Condition 1. *For all infinite executions α of A, $Do(\alpha) \geq 1$ and for all finite executions α of A, there exists an execution fragment α', s.t. $\alpha \cdot \alpha' \in execs(A)$ and $Do(\alpha \cdot \alpha') \geq 1$.*

We proceed with a lemma, which shows that one may construct two executions that contain f failures and their states are indistinguishable to all correct processes, for algorithms that solve the at-most-once problem. Moreover we show that exactly f jobs are performed in the first execution, while no jobs are performed in the second one. Then

we use these executions to prove the main theorem of this section, which shows that the second execution we construct from the lemma, cannot be extended to perform more than $n-f$ tasks. This implies that the effectiveness of any algorithm that solves the at-most-once problem is at most $n-f$.

Lemma 3.1. *If algorithm A solves the at-most-once problem in the presence of $f < m$ crashes and Condition 1 holds, then there exist finite executions $\alpha_1, \alpha_2 \in execs(A)$, s.t. $F_{\alpha_1} = F_{\alpha_2}$, $|F_{\alpha_1}| = |F_{\alpha_2}| = f$, $Do(\alpha_1) = f$, $Do(\alpha_2) = 0$, and the final states of α_1 and α_2 are indistinguishable for all processes in $\mathcal{P} - F_{\alpha_1}$.*

Proof. We prove the lemma by induction on the number of crashes f.

Base case: First we find execution α s.t. $Do(\alpha) \geq 1$ and $F_\alpha = \emptyset$. Such an execution exists by Condition 1 and the fact that crashes are determined by the adversary. Let us consider the first do event in α. Let $\mathsf{do}_{p,j}$ be that event, and let s_1 and s_2 be the states in α before and after $\mathsf{do}_{p,j}$. Since $\mathsf{do}_{p,j}$ does not change shared memory, s_1 and s_2 differ only in the state of process p and thus are indistinguishable for all processes in $\mathcal{P} - \{p\}$. Let $\alpha' = \alpha_0 \cdot (s_1, \mathsf{do}_{p,j}, s_2)$ be the prefix of α up to event $\mathsf{do}_{p,j}$. Clearly $\alpha' \in execs(A)$. We construct the executions $\alpha_1 = \alpha_0 \cdot (s_1, \mathsf{do}_{p,j}, s_2, \mathsf{stop}_p, s_2')$ and $\alpha_2 = \alpha_0 \cdot (s_1, \mathsf{stop}_p, s_1')$. These executions are finite, and since the crashes are controlled by the adversary $\alpha_1, \alpha_2 \in execs(A)$. Moreover $F_{\alpha_1} = F_{\alpha_2} = \{p\}$ and $Do(\alpha_1) = 1$, $Do(\alpha_2) = 0$. Since stop_p affects only the state of p, s_1, s_1', s_2, s_2' are indistinguishable for all processes in $\mathcal{P} - \{p\}$.

Inductive step: For $k < f$ assume that there exist finite executions $\alpha_1, \alpha_2 \in execs(A)$, s.t. $F_{\alpha_1} = F_{\alpha_2}$, $|F_{\alpha_1}| = |F_{\alpha_2}| = k$, $Do(\alpha_1) = k$, $Do(\alpha_2) = 0$ and the final states of α_1 and α_2 are indistinguishable for all processes in $\mathcal{P} - F_{\alpha_1}$. We next construct the needed executions for $k+1$ failures.

We first take α_2. From Condition 1 there exists execution fragment α that has no crashes s.t. $\alpha_2 \cdot \alpha \in execs(A)$ and $Do(\alpha_2 \cdot \alpha) \geq 1$. Since $Do(\alpha_2) = 0$ only α has do

events. Moreover since $\alpha_2 \cdot \alpha \in execs(A)$, α has only actions from processes in $\mathcal{P} - F_{\alpha_2}$. Let $\mathsf{do}_{p,j}$ be the first do event in α, where $p \in \mathcal{P} - F_{\alpha_2}$ and $j \in \mathcal{J}$, and let s_1, s_2 be the states in α before and after $\mathsf{do}_{p,j}$. Clearly s_1 and s_2 are indistinguishable for all processes in $\mathcal{P} - \{p\}$. Let us consider the prefix of $\alpha_2 \cdot \alpha$ up to event $\mathsf{do}_{p,j}$ and let us denote this as $\alpha_2 \cdot \alpha_0 \cdot (s_1, \mathsf{do}_{p,j}, s_2)$. We have that $\alpha_2 \cdot \alpha_0 \cdot (s_1, \mathsf{do}_{p,j}, s_2) \in execs(A)$.

Note that since the final states of α_1 and α_2 are indistinguishable for all processes in $\mathcal{P} - F_{\alpha_2}$, and α_0 contains only actions from process in $\mathcal{P} - F_{\alpha_2}$, the actions of the execution fragment α_0 can extend execution α_1 leading to a state s_3 that is indistinguishable for all processes in $\mathcal{P} - F_{\alpha_2}$ from state s_1. This means that there exists execution fragment α_0' that has the same sequence of actions with α_0, s.t. $\alpha_1 \cdot \alpha_0' \cdot (s_3, \mathsf{do}_{p,j}, s_4) \in execs(A)$ and s_1, s_2, s_3, s_4 are indistinguishable for all processes in $\mathcal{P} - (F_{\alpha_1} \cup \{p\})$. Since $\alpha_1 \cdot \alpha_0' \cdot (s_3, \mathsf{do}_{p,j}, s_4) \in execs(A)$, it must hold that $j \notin J_{\alpha_1}$.

We construct the executions $\alpha_2' = \alpha_2 \cdot \alpha_0 \cdot (s_1, \mathsf{stop}_p, s_1')$ and $\alpha_1' = \alpha_1 \cdot \alpha_0' \cdot (s_3, \mathsf{do}_{p,j}, s_4, \mathsf{stop}_p, s_4')$. We have that $\alpha_1', \alpha_2' \in execs(A)$, $F_{\alpha_1'} = F_{\alpha_2'} = F_{\alpha_1} \cup \{p\}$, $|F_{\alpha_1'}| = k+1$, $Do(\alpha_1') = k+1$, $Do(\alpha_2') = 0$, states s_1', s_4' are indistinguishable for all processes in $\mathcal{P} - F_{\alpha_1'}$. □

Theorem 3.2. *If algorithm A solves the at-most-once problem in the presence of $f < m$ crashes, then there exists an execution $\alpha \in execs(A)$, s.t. either α is infinite and $Do(\alpha) \leq n - f$, or α is finite, and there exists no execution fragment α', s.t. $\alpha \cdot \alpha' \in execs(A)$ and $Do(\alpha \cdot \alpha') > n - f$.*

Proof. By contradiction. Assume the theorem to be false, with Condition 1 holding. Thus from Lemma 3.1 we can construct finite executions $\alpha_1, \alpha_2 \in execs(A)$, s.t. $F_{\alpha_1} = F_{\alpha_2}$, $|F_{\alpha_1}| = |F_{\alpha_2}| = f$, $Do(\alpha_1) = f$, $Do(\alpha_2) = 0$ and the final states of α_1 and α_2 are indistinguishable for all processes in $\mathcal{P} - F_{\alpha_1}$. Also from the assumption, there exists execution fragment α' s.t. $\alpha_2 \cdot \alpha' \in execs(A)$ and $Do(\alpha_2 \cdot \alpha') > n - f$. Since $Do(\alpha_2) = 0$,

it must be that $Do(\alpha') > n - f$. Clearly α' has only actions for processes in $\mathcal{P} - F_{\alpha_2} = \mathcal{P} - F_{\alpha_1}$. Because the final states of α_1 and α_2 are indistinguishable for all processes in $\mathcal{P} - F_{\alpha_1}$ the sequence of actions in α' can extend α_1 as well. This means that there exists execution fragment α'' that has exactly the same actions as α' s.t. $Do(\alpha'') > n - f$ and $\alpha_1 \cdot \alpha'' \in execs(A)$. But $Do(\alpha_1) = f$ and $J_{\alpha_1}, J_{\alpha''} \subseteq \mathcal{J}$. Since $n = |\mathcal{J}|$ it follows by the pigeonhole principle that $J_{\alpha_1} \cap J_{\alpha''} \neq \emptyset$ and thus $\alpha_1 \cdot \alpha''$ is not an at-most-once execution, a contradiction. □

The main result follows as a corollary to the theorem.

Corollary 3.3. *For all algorithms A that solve the at-most-once problem with m processes and $n \geq m$ jobs in the presence of $f < m$ crashes it holds that $E_A(n, m, f) \leq n - f$.*

3.2 Consensus Number for Strong At-Most-Once

In this sections we show that the strong at-most-once problem has consensus number 2. As a result, we have from [23] that there exists no wait-free deterministic algorithm that solves the strong at-most-once problem, using only atomic read/write registers. Current deterministic solutions for the at-most-once, as presented in [29, 27], use only atomic read/write registers and are wait-free, thus they do not offer a solution for the strong at-most-once problem.

We start by proving that the strong at-most-once problem has consensus number at least 2. Then we will prove that the strong at-most-once problem can be solved using test-and-set operations and as a result it cannot have consensus number greater than 2.

Lemma 3.4. *The strong at-most-once problem has consensus number at least* 2.

Proof. In order to prove that the strong at-most-once problem has consensus number at least 2, we will assume that we have a wait-free algorithm A that solves the strong at-most-

once problem for 2 processes, and we will demonstrate how to implement a wait-free algorithm A' for the consensus problem for 2 processes, using algorithm A and atomic read/write shared memory registers. Since the nature of the jobs performed by algorithm A are external to the problem and the algorithm, we treat algorithm A as a black box implementation, and define the jobs algorithm A is performing so that when that process p performs job i, it writes its identifier p in the $i-$th position of an array W. Algorithm A' works for 2 processes p, q and uses an array C of size 2 and an array W of size n where n the size of the strong at-most-once problem algorithm A solves. Array W has its cells initialized to the $\perp$ value. As stated before when a process p performs a job i in algorithm A, process p writes its process identifier at position i of the array W. Since algorithm A solves the at-most-once problem only one process may write its identifier in any single position of the array W. Algorithm A' works as follows:

Process p writes its proposed input value in position p of the array C and then invokes algorithm A. After process p terminates the execution of algorithm A, process p reads the value stored in position 0 of the array W. If the value it reads is its process identifier, it decides the value it proposed, otherwise it decides on the value proposed by the other process that participates in algorithm A'.

Since algorithm A is wait-free algorithm A' is also wait-free. Now we need to prove consistency and validity, namely that distinct processes never decide on distinct values and that the common decision value is the input to some process.

For process p we have 2 cases, process p either decides the value stored in C_p, or the value stored in C_q. Case 1: Process p decides C_p. The value it decides can be only the input value of process p, since from A' only p may write in C_p and process p first writes its input value in C_p, then participates in algorithm A and from the outcome of algorithm A, it either reads C_p or C_q and decides on the value it read. Since process p

reads C_p it follows that it reads in position W_0 its process id, and thus p performed job 0. Moreover if process q participated in A, q did not perform job 0 (at-most-once property). So from A' it follow that q decided on C_p. Now we only need to prove that the value q read in C_p is the input of p. If q read a value different than the input of p, it follows that when q returned from the invocation of algorithm A, p had not written yet its input value in C_p, and consequently had not invoked algorithm A. This is a contradiction, since algorithm A solves the strong at-most-once problem. If process q is the only process invoking algorithm A and q does not fail, then the effectiveness of algorithm A should be n, which implies that process q performed job 0, a contradiction.

Case 2: Process p decides C_q. This means that process p invoked algorithm A, the algorithm returned and process p did not perform job 0. Algorithm A solves the strong at-most-once problem, this means that process q invoked algorithm A before process p terminated (otherwise process p would be executing alone algorithm A and from the strong optimal effectiveness, process p should have performed all the at-most-once jobs), which from algorithm A' implies that C_q contains the input value of process q. Process q either terminates algorithm A without crashing, or crashes before completing algorithm A. In the first case, since process p terminated and did not crash while executing algorithm A, we have that algorithm A should have effectiveness n, which implies that process q executed job 1, and if process q decides, it decides on the value of C_q, the input of q. Otherwise process q has crashed and p is the only process that decides at the input of q. This completes the proof. □

Because atomic read/write registers have consensus number 1 from Lemma 3.4, we have the following Corollary.

Corollary 3.5. *There exists no wait-free deterministic solution for the strong at-most-once problem, using atomic read/write registers.*

Lemma 3.6. *The strong at-most-once problem has consensus number at most* 2.

Proof. We will demonstrate that a wait-free solution for the strong at-most-once problem can be constructed using test-and-set operators. Since test-and-set has consensus number 2, it follows that the strong at-most-once problem cannot have consensus number more than 2.

The algorithm is straight forward. You associate 1 test-and-set operator with each job, which is initially set to 0. Each process p starting from job 1 and moving to job n tests if the operator is 0 and sets it to 1. If the test-and-set operation is successful, process p performs the associated job and moves to the next test-and-set operation. If the test-and-set operation fails, the process just moves to the next test-and-set. It is easy to see that a job will not be performed, only if a process wins the test-and-set and then fails. Since a process after winning a test-and-set performs the associated job, before it attempts to acquire a new test-and-set, a process that fails cannot result in more than one job being lost. This holds since only a process that attempts to aqcuire a test-and-set, may win it. Moreover if at least one process is alive, it will eventually invoke all the test-and-sets and perform all the tasks. Thus the algorithm provided above solves the at-most-once problem with effectiveness $n - f_k$. □

Theorem 3.7. *The strong at-most-once problem has consensus number* 2.

Proof. Theorem 3.7 follows directly from Lemmas 3.4 and 3.6. □

It is easy to see that by associating each at-most-once job with one test-and-set object and having the process that succeeds in the test-and-set perform the at-most-once job before accessing a new test-and-set, we have an effectiveness optimal strong at-most-once solution for an unbounded number of processes using only read/write registers and consensus 2 objects.

Corollary 3.8. *The strong at-most-once problem belongs in the Common*2 *class of objects.*

The *Common*2 class of objects, is the class of objects that contains (1) any read-modify-write object that applies commutative functions, and (2) any read-modify-write-object that applies overwrite functions.

3.3 Open Problems

There is a need for an impossibility result on wait-free deterministic solutions that use atomic read/write registers for the at-most-once problem, similar to the one for the strong at-most-once. We need to explore what constrains on the effectiveness of solutions could give us such an impossibility result. We conjecture that such a result exists if you require optimal effectiveness. Can we get something stronger, proving that even sub-optimal wait-free deterministic solutions cannot be achieved?

We are currently also missing lower bounds on the work of solutions with effectiveness near $n - f$. Establishing first an impossibility result on wait-free deterministic solutions may help in asking the correct questions, as to for what effectiveness should one seek such a lower bound.

Finally lower bounds on the space complexity of algorithms with effectiveness near $n - f$ need to be established.

4 Collision Avoidance Based Solutions

In this section we present asynchronous deterministic algorithms that solve the at-most-once problem. The algorithms are presented and analyzed using the Input Output Automata formalism as presented in the Section 2.

4.1 Two Process Algorithms for At-Most-Once Problem

We present algorithms for the at-most-once problem that use a collision-avoidance approach. First we give 2-process algorithms: $\text{AO}_{2,n}$ that uses n 1-bit shared memory variables, and $\text{AO}'_{2,n}$ that uses two shared memory variables of $\log n$ bits, thus achieving lower space complexity. Both algorithms achieve optimal effectiveness. The two-process algorithms can be used as building blocks to construct algorithms for larger numbers of processes. Here we use algorithm $\text{AO}_{2,n}$ to construct an m-process algorithm for the at-most-once problem.

4.1.1 Algorithm $\text{AO}_{2,n}$

The algorithm, given in Fig. 1, solves the at-most-once problem for n jobs, using two processes, numbered 0 and 1, and n 1-bit shared variables. The main idea is to have the processes move towards each other, with process 0 performing jobs in the ascending order, and process 1 in the descending order. The processes avoid a *collision*, i.e., doing a job twice, by adopting a *"look ahead decide for the current"* (LA-DC) approach.

The algorithm uses n shared bit variables $x_0, \ldots, x_{n-1}$ as a bookkeeping mechanism to record progress. Initially all shared variables are set to 0. If process p performs job j using action $\mathsf{do}_{p,j}$, then $status_p$ variable is changed to *set*. This enables action set_p that in turn sets the value of x_j to 1. The process decides whether a job can be performed in action check_p. Using the LA-DC approach, before a process performs job j, it decides that it is safe to do so, by checking the shared variable associated with the next job in its path, that is x_{j+1} for process 0 and x_{j-1} for process 1. If x_{j+1} (resp. x_{j-1}) is 0 then process 0 (resp. 1) proceeds to perform j; otherwise the status of the process is assigned the value *end*, and we say that the process *terminates*. The key idea is that since x_{j+1} (resp. x_{j-1}) is 0 then the competing process 1 (resp. process 0), did not yet perform the

Shared Variables: $\mathcal{X} = \{x_0, \ldots, x_{n-1}\}$, boolean, initially all 0

Signature:

Input:
stop_p, $p \in \{0,1\}$

Output:
$\mathsf{do}_{p,j}$, $p \in \{0,1\}$, $j \in \mathcal{J}$

Internal:
next_p, $p \in \{0,1\}$
Read: check_p, $p \in \{0,1\}$
Write: set_p, $p \in \{0,1\}$

State:

$status_p \in \{check, set, do, done, end, stopped\}$, initially $check$
$cur_p \in \{0, \ldots, n-1\}$, initially $cur_0 = 0$ and $cur_1 = n-1$
$step_p \in \{-1, 1\}$, initially $step_0 = 1$ and $step_1 = -1$

Transitions of process p:

Internal Read check_p
Precondition:
  $status_p = check$
Effect:
  if $(cur_p + step_p) \geq 0$ AND $(cur_p + step_p) \leq n-1$
  then
    if $x_{cur_p + step_p} = 0$
    then $status_p \leftarrow do$
    else $status_p \leftarrow end$
  else
    $status_p \leftarrow end$

Internal next_p
Precondition:
  $status_p = done$
Effect:
  $cur_p \leftarrow cur_p + step_p$
  $status_p \leftarrow check$

Internal Write set_p
Precondition:
  $status_p = set$
Effect:
  $x_{cur_p} \leftarrow 1$
  $status_p \leftarrow done$

Output $\mathsf{do}_{p,j}$
Precondition:
  $status_p = do$
  $cur_p = j$
Effect:
  $status_p \leftarrow set$

Input stop_p
Effect:
  $status_p \leftarrow stopped$

Figure 1: Algorithm $\text{AO}_{2,n}$: Shared Variables, Signature, States and Transitions

task $j+1$ (resp. $j-1$). Hence it cannot be performing j and collision is avoided.

To show correctness we first prove that if $cur_0 = k$ for some $k > 0$, then all shared variables "before" x_k are set to 1, and respectively that if $cur_1 = k$, then all shared variables "after" x_k are set to 1.

Lemma 4.1. *For any execution α of $\text{AO}_{2,n}$ and for any state s in α such that $s.cur_0 = k$ and $s.cur_1 = k'$ for $1 \leq k \leq k' \leq n-2$, then for $i \in \{0, ..., k-1\} \cup \{k'+1, ..., n-1\}$, $s.x_i = 1$, and actions $\mathsf{do}_{*,i}$ precede s in α.*

Proof. We first prove the claim for process 0 by induction on k.

Base Case: We show that in any execution α of $\text{AO}_{2,n}$, for any state s, s.t. $s.cur_0 = 1$, it holds that $s.x_0 = 1$ and an action $\mathsf{do}_{0,0}$ precedes s in α. If $s.cur_0 = 1$, then from algorithm $\text{AO}_{2,n}$ state s is preceded in α by the transition $(\{status_0 = done, cur_0 = 0\}, next_0, \{status_0 = check, cur_0 = 1\})$.

Since in this transition $status_0 = done$, it must be preceded by the transition $(\{status_0 = set, cur_0 = 0, x_0 = 0\}, set_0, \{status_0 = done, cur_0 = 0, x_0 = 1\})$, which in turn, since $status_0 = set$ must be preceded by the transition $(\{status_0 = do, cur_0 = 0\}, \mathsf{do}_{0,0}, \{status_0 = set, cur_0 = 0\})$. So if $s.cur_0 = 1$, then $s.x_0 = 1$, since state s is preceded by action $\mathsf{do}_{0,0}$ and action set_0 with $cur_0 = 0$.

Induction Hypothesis. Assume that for any execution α of $\text{AO}_{2,n}$, and any state s, s.t. $s.cur_0 = j$ it holds that $\forall i \in 0, \ldots, j-1 : s.x_i = 1$ and $\mathsf{do}_{0,i}$ action precedes s.

Inductive Step: If in execution α of $\text{AO}_{2,n}$, a state s s.t. $s.cur_0 = j+1$ has been reached, the transition $(\{status_0 = done, cur_0 = j\}, next_0, \{status_0 = check, cur_0 = j+1\})$ precedes s. Moreover this transition is preceded by the transition $(\{status_0 = set, cur_0 = j, x_j = 0\}, set_0, (\{status_0 = done, cur_0 = j, x_j = 1\})$, which in turn is preceded by the transition $(\{status_0 = do, cur_0 = j\}, \mathsf{do}_{0,j}, \{status_1 = set, cur_1 = j\})$. From the above we conclude that $s.x_j = 1$ and the action $\mathsf{do}_{0,j}$ precedes state s. Moreover since a state s' with $s'.cur_0 = j$ precedes in α state s, and since algorithm $\text{AO}_{2,n}$ never sets a shared variable from 1 to 0, then by the Induction Hypothesis $\forall i \in 1, \ldots, j-1 : s.x_i = 1$ and $\mathsf{do}_{0,i}$ action precedes s.

With similar arguments we can prove the claim for process 1. □

Using Lemma 4.1 we prove that $\text{AO}_{2,n}$ solves the at-most-once problem.

Theorem 4.2. *Algorithm $\text{AO}_{2,n}$ solves the at-most-once problem.*

Proof. We want to show that algorithm $\text{AO}_{2,n}$ solves the at-most-once problem. To derive contradiction assume that there exists an execution α of $\text{AO}_{2,n}$ that contains two distinct actions $\mathsf{do}_{p,j}$ and $\mathsf{do}_{q,j}$, for $j \in \mathcal{J}$ and $p, q \in \{0,1\}$. We examine the following cases.

- Case 1: $p = q$. This means that job j is performed twice by the same process, process p. W.l.o.g. let $\mathsf{do}_{p,j}$ precede $\mathsf{do}_{q,j}$. The transition for action $\mathsf{do}_{p,j}$ in α

is $(\{status_p = do, cur_p = j\}, \mathsf{do}_{p,j}, \{status_p = set, cur_p = j\})$. This can happen only once in the execution α for process p, since according to algorithm $\text{AO}_{2,n}$, the $status_p$ internal variable becomes do again only if the set_p, next_p and check_p actions occur in that order. But next_p, increments(if $p = 0$), or decrements(if $p = 1$) the variable cur_p. So for any subsequent state s' with $s'.status_p = do$, $s'.cur_p = j'$ it must hold that either $j' > j$ (if $p = 0$) or $j' < j$ (if $p = 1$). Thus case 1 is impossible.

- Case 2: $p \neq q$ and $j = 0$. Both a $\mathsf{do}_{0,0}$ and a $\mathsf{do}_{1,0}$ exist in α. The transition of action $\mathsf{do}_{1,0}$, is $(\{status_1 = do, cur_1 = 0\}, \mathsf{do}_{1,j}, \{status_1 = set, cur_1 = 0\})$. A state s s.t. $s.status_1 = do$ and $s.cur_1 = 0$ must be preceded in α by the transition $(\{status_1 = check, cur_1 = 0\}, \mathsf{check}_1, \{status_1 = do, cur_1 = 0\})$. This is a contradiction since a check_1 action from a state with $\{status_1 = check, cur_1 = 0\}$, will lead to a state with $\{status_1 = end, cur_1 = 0\}$ from the first **if** clause of check_1 action. After reaching a state with $status_1 = end$ process 1 stops performing any locally controlled actions.

- Case 3: $p \neq q$ and $j = n-1$. Both a $\mathsf{do}_{0,n-1}$ and a $\mathsf{do}_{1,n-1}$ exist in α. The transition of action $\mathsf{do}_{0,n-1}$, is $(\{status_0 = do, cur_0 = n-1\}, \mathsf{do}_{0,n-1}, \{status_0 = set, cur_0 = n-1\})$. A state s s.t. $s.status_0 = do$ and $s.cur_0 = n-1$ must be preceded in α by the transition $(\{status_0 = check, cur_0 = n-1\}, \mathsf{check}_0, \{status_0 = do, cur_0 = n-1\})$. This is a contradiction since a check_0 action from a state with $\{status_0 = check, cur_0 = n-1\}$, will lead to a state with $\{status_0 = end, cur_0 = n-1\}$ from the first **if** clause of check_0 action. After reaching a state with $status_0 = end$ process 0 stops performing any locally controlled actions.

- Case 4: $p \neq q$ and $j \in \{1, \ldots, n-2\}$. Without loss of generality, let $p = 0$ and $q = 1$ (similar arguments can prove the inverse case). The transitions of the two actions are $(\{status_0 = do, cur_0 = j\}, \mathsf{do}_{0,j}, \{status_0 = set, cur_0 = j\})$ and

$(\{status_1 = do, cur_1 = j\}, \mathsf{do}_{1,j}, \{status_1 = set, cur_1 = j\})$. Moreover in α, transition $e_0 = (\{status_0 = check, cur_0 = j, x_{j+1} = 0\}, \mathsf{check}_0, \{status_0 = do, cur_0 = j, x_{j+1} = 0\})$ must precede all the states of α with $\{status_0 = do, cur_0 = j\}$ and transition $e_1 = (\{status_1 = check, cur_1 = j, x_{j-1} = 0\}, \mathsf{check}_1, \{status_1 = do, cur_1 = j, x_{j-1} = 0\})$ must precede all states of α with $\{status_1 = do, cur_1 = j\}$. Without loss of generality, let e_0 precede e_1.Thus a state s with $s.cur_0 = j$, precedes transition e_1 in α. This is a contradiction from the fact that x_{j-1} never changes from 1 to 0 in any execution of $\text{AO}_{2,n}$ and since from Lemma 4.1 we have that if $s.cur_0 = j$, then $s.x_{j-1} = 1$. A similar argument applies if e_1 precedes e_0 in α.

Since all cases are impossible then algorithm $\text{AO}_{2,n}$ solves the at-most-once problem.. □

4.1.2 Algorithm $\text{AO}'_{2,n}$

This algorithm, also uses the LA-DC idea. The difference is that we use two integer shared variables, x_{left} and x_{right}, each of $\log n$ bits, that serve as pointers to the progress of each process. Initially x_{left} and x_{right} are set to 0 and $n-1$ respectively, and thereafter each time process 0 or 1 performs a job with action $\mathsf{do}_{*,*}$, x_{left} is incremented or x_{right} is decremented respectively at event set. The decision (made in action check) on whether it is safe to perform a job is based on the differences $x_{right} - cur_0$ and $cur_1 - x_{left}$ for processes 0 and 1 respectively. If the difference is greater than 1, then it is safe to perform the job. With similar arguments as in Theorem 4.2 the result follows.

Theorem 4.3. *Algorithm* $\text{AO}'_{2,n}$ *solves the at-most-once problem.*

Proof. We proceed similarly as in the proof of Theorem 4.2. In order to get a contradiction we assume that there exists execution α of algorithm $\text{AO}'_{2,n}$ s.t. α is not an at-most-once execution. It follows that α contains two distinct actions $\mathsf{do}_{p,j}$ and $\mathsf{do}_{q,j}$, where $p, q \in \{0,1\}$ for the same job $j \in \mathcal{J}$. We examine the following cases.

- Case 1: $p = q$. This means that the same process performs job j twice. W.l.o.g. let $\mathsf{do}_{p,j}$ precede $\mathsf{do}_{q,j}$. According to algorithm $\text{AO}'_{2,n}$, such action $\mathsf{do}_{p,j}$ occurs only if $cur_p = j$ and $status_p = do$. Action $\mathsf{do}_{p,j}$ set $status_p = set$. The $status_p$ variable becomes do again only if the set_p, next_p and check_p actions occur in that order. Let $p = 0$, after the aforementioned sequence of actions the cur_0 internal variable is incremented from j to $j+1$ (from action next_0). Since cur_0 never decrements, $\mathsf{do}_{0,j}$ can never occur again. With similar arguments we can show that if $p = 1$ action $\mathsf{do}_{1,j}$ can occur only once. Thus case 1 is impossible.

- Case 2: $p \neq q$ and $j = 0$. Both a $\mathsf{do}_{0,0}$ and a $\mathsf{do}_{1,0}$ exist in α. The transition for action $\mathsf{do}_{1,0}$ is $(\{status_1 = do, cur_1 = 0\}, \mathsf{do}_{1,j}, \{status_1 = set, cur_1 = 0\})$. A state s s.t. $s.status_1 = do$ and $s.cur_1 = 0$ must be preceded in α by the transition $(\{status_1 = check, cur_1 = 0\}, \mathsf{check}_1, \{status_1 = do, cur_1 = 0\})$. This transition must in turn be preceded by the transition $(\{status_1 = done, cur_1 = 1\}, \mathsf{next}_1, \{status_1 = check, cur_1 = 0\})$. This is a contradiction since a next_1 action from a state with $\{status_1 = done, cur_1 = 1\}$, will lead to a state with $\{status_1 = end, cur_1 = 0\}$ from the first **if** clause of next_1 action. After reaching a state with $status_1 = end$ process 1 stops performing any locally controlled actions.

- Case 3: $p \neq q$ and $j = n-1$. Both a $\mathsf{do}_{0,n-1}$ and a $\mathsf{do}_{1,n-1}$ exist in α. The transition of action $\mathsf{do}_{0,n-1}$, is $(\{status_0 = do, cur_0 = n-1\}, \mathsf{do}_{0,n-1}, \{status_0 = set, cur_0 = n-1\})$. A state s s.t. $s.status_0 = do$ and $s.cur_0 = n-1$ must be preceded in α by the transition $(\{status_0 = check, cur_0 = n-1\}, \mathsf{check}_0, \{status_0 = do, cur_0 = n-1\})$. This transition must in turn be preceded by the transition $(\{status_0 = done, cur_0 = n-2\}, \mathsf{next}_0, \{status_0 = check, cur_1 = n-1\})$. This is a contradiction since a next_0 action from a state with $\{status_0 = done, cur_0 = n-2\}$, will lead to a state with $\{status_0 = end, cur_0 = n-1\}$ from the first **if** clause of next_0 action. After reaching

a state with $status_0 = end$ process 0 stops performing any locally controlled actions.

- Case 4: $p \neq q$ and $j \in \{1, \ldots, n-2\}$. Without loss of generality, let p=0 and q=1. Actions $\mathsf{do}_{0,j}$ and $\mathsf{do}_{1,j}$ must be preceded by actions check_0 and check_1 respectively. Since for process p, check_p does not modify cur_p or any of the shared variables, x_{left} and x_{right}, it follows that if during $\mathsf{do}_{p,j}$, $cur_p = j$, it must be the case that in the check_p that precedes the do action $cur_p = j$. So in both check_0 and check_1 actions $cur_0 = j$ and $cur_1 = j$ respectively. There are two possibilities to consider: a) check_0 appears before check_1 in the execution trace or b) vice-versa. The cases are symmetrical so we will examine the case where check_0 appears before check_1. As mentioned before during check_0, the value of $cur_0 = j$. Moreover it is easy to see from $\text{AO}'_{2,n}$ that for any execution α and for any state s, $s.x_{left} = s.cur_0 - 1$ or $s.x_{left} = s.cur_0$, so it is the case that for all states that succeed action check_0 $x_{left} \geq j-1$. Similarly for all states that succeed check_1 $x_{right} \leq j+1$. Since check_1 appears after check_0 then $x_{left} \geq j-1$ during check_1 and hence the condition **if** $cur_1 - x_{left} > 1$ is false. Thus the $status_1$ variable never becomes do and the $\mathsf{do}_{1,j}$ is never performed contradicting our assumption. Similarly we can show that the $\mathsf{do}_{0,j}$ is never performed if check_0 appears after check_1 in an execution. Hence this case is also impossible.

Since all cases lead to a contradiction then algorithm $\text{AO}'_{2,n}$ solves the at-most-once problem. □

4.1.3 Effectiveness, Work and Space Complexity

We now present the efficiency results for both algorithms.

Effectiveness: We show that algorithms $\text{AO}_{2,n}$ and $\text{AO}'_{2,n}$ perform $n-1$ jobs in the presence of at most one stopping failure (optimal given Corollary 3.3).

Theorem 4.4. *The effectiveness of* $\text{AO}_{2,n}$ *with* $f < 2$ *is* $E_{\text{AO}_{2,n}}(n,2,f) = n-1$.

Proof. We examine 3 cases, according to which process, if any, fails during an execution.

- Case 1: No process fails.

  In this case a fair execution of $\text{AO}_{2,n}$, is a finite execution $\alpha \in fairexecs(\text{AO}_{2,n})$, where in the final state, $status_p = end$ for both process 0 and process 1. Since the $status_p$ can be set to *end* only through a check_p, transition e_1=$(\{status_0 = check, cur_0 = k, x_{k+1} = 1\}, check_0, (\{status_0 = end, cur_0 = k, x_{k+1} = 1\})$ where $0 \leq k < n-1$ or e_2=$(\{status_1 = check, cur_1 = k', x_{k'-1} = 1\}, check_1, (\{status_1 = end, cur_1 = k', x_{k'-1} = 1\})$ where $0 < k' \leq n-1$, or both must be in α. Let's assume that e_1 is in α. Let s be the enabling state of e_1. We have that $s.x_{k+1} = 1$, thus x_{k+1} must have been set to 1 by process 1 before s. So there exists a transition e_3, that precedes e_1, s.t. e_3=$(\{status_1 = set, cur_1 = k+1, x_{k+1} = 0\}, set_1, \{status_1 = done, cur_2 = k+1, x_{k+1} = 1\})$, which is in turn preceded by transition e_4=$(\{status_1 = do, cur_1 = k+1\}, \mathsf{do}_{1,k+1}, \{status_1 = set, cur_1 = k+1\})$. So at s, $s.cur_0 = k$, $s.cur_1 \leq k+1$ since e_3 precedes e_1 and cur_1 never increases in $\text{AO}_{2,n}$ and the action $\mathsf{do}_{1,k+1}$ has occurred. From Lemma 4.1 we have that, $\forall i \in \{0,\ldots,k-1\}$, an action $\mathsf{do}_{0,i}$ precedes s, $\forall j \in \{k+2,\ldots,n-1\}$, an action $\mathsf{do}_{1,j}$ precedes s, also action $\mathsf{do}_{1,k+1}$ precedes s. This means that in execution α at least $n-1$ jobs where performed. We can use similar arguments for e_2. Normally both e_1 and e_2 are in α. The only case when e_1 is not in α is if process 0 does $n-1$ jobs and cur_0 becomes $n-1$, where e_2 must exist in α, and e_2=$(\{status_1 = check, cur_1 = n-1, x_{n-2} = 1\}, check_1, \{status_1 = end, cur_1 = n-1, x_{n-2} = 1\})$. Similarly, the only case when e_2 is not in α is if process 1 does $n-1$ jobs and cur_1 becomes 1, where e_1 must exist in α, and e_1=$(\{status_0 = check, cur_0 = 0, x_1 = 1\}, check_0, \{status_0 = end, cur_0 = 1, x_1 = 1\})$.

- Case 2: Process 0 stop failed

  In this case a fair execution of $\text{AO}_{2,n}$, is a finite execution $\alpha \in fairexecs(\text{AO}_{2,n})$, where in the final state, $status_1 = end$. Since the $status_1$ can be set to *end* only through a check_1 action either transition e_2=$(\{status_1 = check, cur_1 = k', x_{k'-1} = 1\}, check_1, \{status_1 = end, cur_1 = k', x_{k'-1} = 1\})$, is in α, or in α process 1 does $n-1$ jobs, cur_1 becomes 0 and process 1 performs the transition $(\{status_1 = check, cur_1 = 0\}, check_1, \{status_1 = end, cur_1 = 0\})$. In case that transition e_2 happened, using similar arguments with the previous case we can prove that at least $n-1$ action $\mathsf{do}_{p,j}$ where $p \in 0,1$ and $j \in 0,\ldots,n-1$ are performed.

- Case 3: Process 1 stop failed

  We can use similar arguments as in case 2 to prove that at least $n-1$ $\mathsf{do}_{p,j}$ actions are performed.

Algorithm $\text{AO}_{2,n}$ has no infinite fair executions with at most one stopping failure, since in finite steps either process 0 will reach a state were $status_0 = end$ and process 1 will have stop-failed or process 1 will reach a state were $status_1 = end$ and process 0 will have stop-failed or both processes 0 and 1 will reach a state where $status_0 = end$ and $status_1 = end$. In all theses cases no locally controlled actions are enabled. So an infinite execution should have infinite input actions and the only input actions the automaton describing algorithm $\text{AO}_{2,n}$ has are stop_p actions. So we have that all fair executions of algorithm $\text{AO}_{2,n}$ with at most one stopping failure perform at least $n-1$ tasks and thus $E_{\text{AO}_{2,n}}(n,2,1) = min_{\alpha \in fairexecs_1(\text{AO}_{2,n})}(Do(\alpha)) = n-1$. □

Theorem 4.5. *The effectiveness of* $\text{AO}'_{2,n}$ *with* $f < 2$ *is* $E_{\text{AO}'_{2,n}}(n,2,f) = n-1$.

Proof. We can use similar argument with the proof of Theorem 4.4. The main difference is that instead of using Lemma 4.1, we use the fact that from the construction of $\text{AO}'_{2,n}$,

if in a state s, $s.cur_0 = k$, then $s.x_{left} \in \{k-1, k\}$ and $\forall i \in \{0, \ldots, s.x_{left}\}$, an action $\mathsf{do}_{0,i}$ precedes s, and similarly if $s.cur_1 = k'$, then $s.x_{right} \in \{k', k'+1\}$ and $\forall j \in \{s.x_{right}, \ldots, n-1\}$, an action $\mathsf{do}_{1,j}$ precedes s. □

Work and Space: Next we asses the work and space complexity of algorithms $\text{AO}_{2,n}$ and $\text{AO}'_{2,n}$. For the work complexity we count the total number of bits accessed during memory accesses. For space complexity we count the total number of bits used in shared and local memory. Recall that algorithm $\text{AO}_{2,n}$ uses single bit shared variables and $\text{AO}'_{2,n}$ uses shared variables of $\log n$ bits.

Theorem 4.6. *Algorithm* $\text{AO}_{2,n}$ *has work* $2(n+1)$ *and space* $n + 2\log n + 8$ *bits.*

Proof. From Theorem 4.2, $\text{AO}_{2,n}$ performs at most n writes. Preceding each write, a process performs a read to decide whether it is safe to perform a job. Each process performs one additional read operation, corresponding to termination. Thus there are n writes and $n+2$ reads. The algorithm uses n shared 1-bit variables, 2 internal variables (one for each process) of $\log n$ bits (cur_p), 2 internal variables of 3 bits ($status_p$), and 2 internal variables of 1 bit ($step_p$). □

Theorem 4.7. *Algorithm* $\text{AO}'_{2,n}$ *has work* $2(n+1)\log n$ *and space* $4\log n + 10$ *bits.*

Proof. Algorithm $\text{AO}'_{2,n}$ performs n writes and $n+2$ reads. Each read and each write accesses $\log n$ bits. The algorithm uses 2 shared variables of $\log n$ bits, 2 internal variables of $\log n$ bits (cur_p), 2 internal variables of 3 bits ($status_p$), 2 internal variables of 1 bit ($step_p$), and 2 internal variables of 1 bit (pid_p). □

4.2 Collision Avoidance Multiprocess Solution for the At-Most-Once problem

4.2.1 Multiprocess Algorithm $\text{AO}_{m,n}$

We now present m-process algorithm $\text{AO}_{m,n}$, given in Fig. 2, where $m = 2^h$, and the number of jobs is $n = k^h$ (non-powers are handled using standard padding techniques). The algorithm is a hierarchical generalization of algorithm $\text{AO}_{2,n}$. It uses an abstract full k-ary tree of h levels to keep track of progress and guarantee at-most-once semantics. All processes start at the root of the tree at level 0. At each node λ at level μ processes are split in two groups according to their process identifiers and look for subtrees with jobs that are safe to perform in the children of node λ. Thus at each node λ we can see the processes as two groups, group 0 and group 1, solving a sub-problem with k groups of jobs (the subtrees rooted at the children of node λ) using the approach of algorithm $\text{AO}_{2,n}$. Group 0 starts from the leftmost child of node λ and moves to the right, while group 1 starts from the rightmost child and moves to the left. Both groups use the LA-DC approach to define whether it is safe to perform a group of jobs (sub-tree rooted at a child of node λ).

We store the tree on a shared memory array by associating each node with a shared variable. Variable x_0 is associated with the root at level 0, $x_1,\ldots,x_k$ with the nodes at level 1, $x_{k+1},\ldots,x_{k^2}$ with the nodes at level 2, and so on. In general the nodes at level $\mu \in [1\ldots h]$ are associated with the shared variables $x_{u_\mu},\ldots,x_{u_\mu+k^\mu-1}$, where $u_\mu = 1+k+k^2+k^3+\ldots+k^{\mu-1}$. The tree has a total of $v = u_{h+1}$ nodes. We denote by node λ the node associated with the shared variable x_λ, that has children associated with $x_{\lambda\cdot k+1},\ldots,x_{\lambda\cdot k+k}$ and a parent associated with $x_{\lfloor\frac{\lambda-1}{k}\rfloor}$. Node $\lambda \in [0\ldots v-1]$ is at level $\mu = \lfloor\log_k(\lambda\cdot(k-1)+1)\rfloor$. Finally, job j is associated with leaf x_{u_h+j}. Next we present $\text{AO}_{m,n}$ in more detail.

Internal Variables of process p

$status_p \in \{check, set, up, down, do, done, end, stopped\}$ records the status of process p and defines its next action as follows: *down–p* can move to the children of its current node, *up–p* finished the current level and can move one level higher, *set–p* can set the shared variable associated with its current node to 1, *check–p* has to check whether it is safe to work at the current node, *do–p* is at a leaf and can perform the associated job, *done–p* finished working at the current node and can move to the next, *end–p* terminated (it is not safe for p to work on the tree), *stopped–p* crashed. All processes start at node 0, with $status_p = down$.

$pid_p[0 \ldots h]$ is a binary expansion of p into $h+1$ bits. Note that $p \in [0, 2^h - 1]$.

$cur_p \in \{0, \ldots, v-1\}$ marks the node at which process p is positioned.

$left_p, right_p \in \{0, \ldots, v-1\}$ keeps the leftmost and rightmost siblings of the current node.

$lvl_p \in \{0, \ldots, h\}$ stores the level μ of the current node.

$step_p \in \{-1, 1\}$ tracks of whether process p is moving from right to left or left to right at the current level.

Actions of process p

down_p: Process p moves one level down. If a leaf is reached, it sets $status_p = do$ in order for the job associated with the leaf to be performed. If p is at an internal node, it checks whether $pid_p[lvl_p]$ is 0 or 1. If it is 0, then p moves to the leftmost child of node cur_p, otherwise it moves to the rightmost child. Process p sets lvl_p, cur_p, $left_p$, $right_p$ and $step_p$ accordingly. The status of p remains *down*.

check_p: If p works left-to-right and cur_p is the rightmost child of its parent, it sets $status_p = up$. Similarly if p works right-to-left and cur_p is the leftmost child of its parent, it sets $status_p = up$. Otherwise, p performs a look-ahead read in shared memory to

determine if it is safe to work on the subtree rooted at node cur_p. If the shared variable associated with the next node ($cur_p + step_p$) is 0, it is safe to work on the subtree of node cur_p and thus sets $status_p = down$. Otherwise it sets $status_p = up$.

up_p: Process p moves one level up. If it is at level 1 (only root is above), it sets $status_p = end$ and terminates. If by moving up an internal node is reached, p updates its internal variables accordingly by checking the proper bit of its pid_p variable, and sets $status_p = set$.

set_p: Process p writes 1 to the shared variable associated with the node cur_p and sets $status_p = done$.

next_p: Process p moves to the next node (left or right, per value of $step_p$), and sets $status_p = check$.

$\mathsf{do}_{p,j}$: Process p preforms job j. Then p sets $status_p = set$.

stop_p: Process p crashes by setting $status_p = stopped$.

4.2.2 Correctness and Effectiveness

We show that algorithm $\text{AO}_{m,n}$ solves the at-most-once problem. First we prove that at any internal node λ at level μ, either only processes with $pid_p[\mu] = 0$, or only processes with $pid_p[\mu] = 1$ enter the subtree rooted at λ.

Lemma 4.8. *For any execution α of algorithm $\text{AO}_{m,n}$ if there exist states s, s' in α and processes $p, q \in \mathcal{P}$ s.t. $\left\lfloor \frac{s.cur_p - 1}{k} \right\rfloor = \left\lfloor \frac{s'.cur_q - 1}{k} \right\rfloor = \lambda$, for some node λ at level μ, then $pid_p[\mu] = pid_q[\mu]$.*

Proof. For node λ at level μ, if it is the leftmost child of its parent, then from the first **if** clause of action check_p, only processes with $pid_p[\mu] = 0$ may enter the subtree rooted at λ. Similarly if node λ is the rightmost child, only processes with

Shared Variables: $\mathcal{X} = \{x_0, \ldots, x_{v-1}\}$, x_i boolean initially 0

Signature:

Input:	Output:	Internal:	
stop_p, $p \in \mathcal{P}$	$\mathsf{do}_{p,j}$, $p \in \mathcal{P}$, $j \in \mathcal{J}$	next_p, $p \in \mathcal{P}$	*Read:* check_p, $p \in \mathcal{P}$
		up_p, $p \in \mathcal{P}$	*Write:* set_p, $p \in \mathcal{P}$
		down_p, $p \in \mathcal{P}$	

State:

$status_p \in \{check, set, up, down, do, done, end, stopped\}$, initially $down$

$pid_p[0 \ldots h]$, where $pid_p[i] = \left\lfloor \frac{p}{2^{h-i}} \right\rfloor \mod 2$ (the binary expansion of p to $h+1$ bits)

$cur_p \in \{0, \ldots, v-1\}$, initially 0
$left_p \in \{0, \ldots, v-1\}$, initially 0
$right_p \in \{0, \ldots, v-1\}$, initially 0
$lvl_p \in \{0, \ldots, h\}$, initially 0
$step_p \in \{-1, 1\}$, initially undefined

Transitions of process p:

Input stop_p
Effect:
  $status_p \leftarrow stopped$

Internal Read check_p
Precondition:
  $status_p = check$
Effect:
  if $(cur_p + step_p) \geq left_p$
  AND $(cur_p + step_p) \leq right_p$
  then
    if $x_{cur_p + step_p} = 0$
    then
      $status_p \leftarrow down$
    else $status_p \leftarrow up$
  else
    $status_p \leftarrow up$

Internal next_p
Precondition:
  $status_p = done$
Effect:
  $cur_p \leftarrow cur_p + step_p$
  $status_p \leftarrow check$

Internal up_p
Precondition:
  $status_p = up$
Effect:
  if $lvl_p = 1$ **then**
    $status_p \leftarrow end$
  else
    $lvl_p \leftarrow lvl_p - 1$
    $cur_p \leftarrow \left\lfloor \frac{cur_p - 1}{k} \right\rfloor$
    $left_p \leftarrow \left\lfloor \frac{cur_p - 1}{k} \right\rfloor \cdot k + 1$
    $right_p \leftarrow \left\lfloor \frac{cur_p - 1}{k} \right\rfloor \cdot k + k$
    if $pid_p[lvl_p] = 0$ **then**
      $step_p \leftarrow 1$
    else
      $step_p \leftarrow -1$
    $status_p \leftarrow set$

Internal Write set_p
Precondition:
  $status_p = set$
Effect:
  $x_{cur_p} \leftarrow 1$
  $status_p \leftarrow done$

Internal down_p
Precondition:
  $status_p = down$
Effect:
  if $lvl_p = h$ **then**
    $status_p \leftarrow do$
  else
    $lvl_p \leftarrow lvl_p + 1$
    $left_p \leftarrow cur_p \cdot k + 1$
    $right_p \leftarrow cur_p \cdot k + k$
    if $pid_{lvl_p} = 0$ **then**
      $cur_p \leftarrow left_p$
      $step_p \leftarrow 1$
    else
      $cur_p \leftarrow right_p$
      $step_p \leftarrow -1$

Output $\mathsf{do}_{p,j}$
Precondition:
  $status_p = do$
  $cur_p = u_h + j$
Effect:
  $status_p \leftarrow set$

Figure 2: Algorithm $\text{AO}_{m,n}$: Shared Variables, Signature, States and Transitions

$pid_p[\mu] = 1$ may enter the subtree rooted at λ. If node λ is between the leftmost and rightmost children of its parent$\left(\lambda \in \left[\left\lfloor \frac{\lambda-1}{k} \right\rfloor \cdot k + 2 \ldots \left\lfloor \frac{\lambda-1}{k} \right\rfloor \cdot k + k - 1\right]\right)$, then processes with $pid_p[\mu] = 0$ will approach it from the left, while processes with $pid_p[\mu] = 1$ will approach it from the right. In order to get a contradiction let us assume that there exists execution α that has states s, s' and processes p, q with $pid_p[\mu] = 0$ and $pid_q[\mu] = 1$, s.t. $\lfloor \frac{s.cur_p - 1}{k} \rfloor = \lfloor \frac{s'.cur_q - 1}{k} \rfloor = \lambda$. This means that both processes have entered the subtree rooted at node λ. For this to happen, there exist in

α transitions $\left(\left\{cur_p = \lambda, status_p = check\right\}, \mathsf{check}_p, \left\{cur_p = \lambda, status_p = down\right\}\right)$ and $\left(\left\{cur_q = \lambda, status_q = check\right\}, \mathsf{check}_q, \left\{cur_q = \lambda, status_q = down\right\}\right)$, that precede s and s' respectively. Recall that p moves left-to-right and q right-to-left, and before moving to a new node at a level, they set the shared variable associated with the previous node to 1. Hence it follows that either $x_{\lambda+1} = 1$ when action check_p took place or $x_{\lambda-1} = 1$ when action $check_q$ took place. If the first case is true, then the state of p becomes $\left\{cur_p = \lambda, status_p = up\right\}$ preventing p from entering the subtree rooted at λ. Otherwise the state of q becomes $\left\{cur_q = \lambda, status_q = up\right\}$ and q never enters the subtree rooted at λ. So it cannot be the case that both process p and q entered the subtree rooted at node λ in α and that completes the proof. □

Lemma 4.9. *For any execution α of algorithm $\text{AO}_{m,n}$ if there exist states s, s' in α and processes $p, q \in \mathcal{P}$ s.t. $\left\lfloor \frac{s.cur_p - 1}{k} \right\rfloor = \left\lfloor \frac{s'.cur_q - 1}{k} \right\rfloor = \lambda$, for some node λ at level μ, then $pid_p[0 \dots \mu] = pid_q[0 \dots \mu]$.*

Proof. We prove this by induction on the level μ of node λ.

Base Case: Here we consider level $\mu = 0$, meaning that all processes that reach the children of the root (node 0) have the same $pid_*[0]$ bit. This holds since $\forall p \in \mathcal{P}, pid_p[0] = 0$. Thus for any execution α of $\text{AO}_{m,n}$, if there exists state s in α s.t. $\left\lfloor \frac{s.cur_p - 1}{k} \right\rfloor = 0$ for some process $p \in \mathcal{P}$, $pid_p[0] = 0$.

Induction Hypothesis: Assume that for any execution α if there exist states s, s' and processes p, q s.t. $\lfloor \frac{s.cur_p - 1}{k} \rfloor = \lfloor \frac{s'.cur_q - 1}{k} \rfloor = \lambda$, for all nodes $\lambda \in [u_\mu \dots u_\mu + k^\mu - 1]$ at level μ, then $pid_p[0...\mu] = pid_q[0...\mu]$.

Induction Step: By Lemma 4.8 we show that $\forall \lambda \in [u_{\mu+1} \dots u_{\mu+1} + k^{\mu+1} - 1]$ at level $\mu + 1$, for any execution α, if there exist states s, s' and processes p, q s.t. $\lfloor \frac{s.cur_p - 1}{k} \rfloor = \lfloor \frac{s'.cur_q - 1}{k} \rfloor = \lambda$, then $pid_p[0 \dots \mu + 1] = pid_q[0 \dots \mu + 1]$. □

From Lemma 4.9 we get Corollary 4.10 that says, that in any execution α of $\text{AO}_{m,n}$,

only one process p, if any, may reach the decision to perform job j associated with leaf $u_\mu + j$. This decision is reflected in α by a state s, where $s.cur_p = u_\mu + j, s.status_p = do$.

Corollary 4.10. *For any execution α of algorithm $\text{AO}_{m,n}$ if there exist states s, s' and processes p, q s.t. $s.cur_p = \lambda, s.status_p = do$ and $s'.cur_q = \lambda, s'.status_p = do$, for some leaf $\lambda \in [u_h \ldots u_h + k^h - 1]$, then $p = q$.*

Theorem 4.11. *Algorithm $\text{AO}_{m,n}$ solves the at-most-once problem.*

Proof. In any execution, action $\mathsf{do}_{p,j}$ occurs only if preconditions $cur_p = u_h + j$ and $status_p = do$ hold. From Corollary 4.10 only one process may reach such state, thus only one $\mathsf{do}_{*,j}$ event may take place for j. □

4.2.3 Work and Space Complexity

Next we assess work and space of algorithm $\text{AO}_{m,n}$. According to the algorithm specification, only the actions check_p and set_p perform memory accesses, and every time they do so, they access exactly one bit.

Theorem 4.12. *The work complexity of algorithm $\text{AO}_{m,n}$ is $O(n + m \log m)$.*

Proof. We observe that for each subtree rooted at an internal node λ at level μ we have a *sub-instance* of the problem for $k^{h-\mu}$ jobs and $2^{h-\mu}$ processes. All processes of such sub-instance have the same prefix at the first μ bits of their *pid* from Lemma 4.9. Let W_μ be an upper bound on work of the sub-instance. Now we consider the first level of the subtree. Processes are split in groups 0 and 1 (with $2^{h-(\mu+1)}$ processes each), according to the value of their $pid_*[\mu + 1]$. Group 0 starts at the leftmost child, group 1 at the rightmost child, and they move towards each other. From Lemma 4.9 we have that only one of the groups, if any, will continue to the sub-instance of the next level, thus we have at most k sub-instances derived at level $\mu + 1$. From algorithm $\text{AO}_{m,n}$, we have that before a process

enters a node, it does a look ahead memory read, and when it leaves a node, it sets the shared variable associated with the node to 1. This means that we have a total of $k+2$ reads and k writes from the two groups. Since each group has $2^{h-(\mu+1)}$ processes, we get $(k+2)\cdot 2^{h-(\mu+1)}$ reads and $k\cdot 2^{h-(\mu+1)}$ writes. From the above discussion we have the following recurrence relation: $W_\mu = k\cdot W_{\mu+1} + (2k+2)\cdot 2^{h-(\mu+1)}$.

Also for level h (k jobs and 2 processes), we have $k+2$ reads and k writes by Theorem 4.6, thus: $W_h = 2k+2$. Combining the above we get:

$W_0 = k\cdot W_1 + (2k+2)\cdot 2^{h-1} = (2k+2)\cdot 2^{h-1}\cdot \sum_{i=0}^{h-1}\left(\frac{k}{2}\right)^i$.

Case $k=2$: $(2k+2)\cdot 2^{h-1}\cdot \sum_{i=0}^{h-1}\left(\frac{k}{2}\right)^i = 6\cdot 2^{h-1}\cdot h = 5m\log m$

Case $k>2$: $(2k+2)\cdot 2^{h-1}\cdot \sum_{i=0}^{h-1}\left(\frac{k}{2}\right)^i = (2k+2)\cdot 2^{h-1}\cdot \frac{\left(\frac{k}{2}\right)^h - 1}{\frac{k}{2}-1} = \frac{2k+2}{k-2}\cdot(n-m) \leq 8(n-m)$, where the penultimate relation follows form $m = 2^h, n = k^h$.

We conclude that $W_0 = \Theta(n+m\log m)$. □

Theorem 4.13. *The space complexity of algorithm* $\text{AO}_{m,n}$ *is* $\Theta(n+m\log n)$.

Proof. From Figure 2: The number of shared bits is $v < 2n$. Each of m processes uses a constant number of internal variables of size $\lceil \log v \rceil \leq 1+\log n$. □

Effectiveness: We now assess the effectiveness of algorithm $\text{AO}_{m,n}$.

Theorem 4.14. *Algorithm* $\text{AO}_{m,n}$ *has effectiveness* $E_{\text{AO}_{m,n}}(n,m,m-1) = (n^{\frac{1}{\log m}} - 1)^{\log m} = n - \log m\cdot o(n)$.

Proof. We observe that for each subtree rooted at an internal node λ at level μ we have a sub-instance of the problem for $k^{h-\mu}$ jobs and $2^{h-\mu}$ processes. Moreover if we consider only the first level of such a sub-instance, we have to solve a problem of k groups of jobs (with $k^{h-(\mu+1)}$ jobs each) and 2 groups of processes (with $2^{h-(\mu+1)}$ processes each). Furthermore, as we pointed out before, algorithm $\text{AO}_{m,n}$ follows the same principles for solving this instance as algorithm $\text{AO}_{2,n}$. Thus at each level we match the effectiveness

of $\text{AO}_{2,n}$ that by Theorem 4.4 performs $E_{\text{AO}_{2,n}}(k,2,1) = k-1$ jobs. If we go all the way down to level $h = \log_k n$, we have an exact instance of the 2-process problem (Section 4.1.1) and hence by Theorem 4.4 it follows that $E_{\text{AO}_{m,n}}(k,2,1) = E_{\text{AO}_{2,n}}(k,2,1) = k-1$. From the above we get the following recurrence:

$$E_{\text{AO}_{m,n}}(n,m,m-1) = (k-1)\cdot E_{\text{AO}_{m,n}}\left(\tfrac{n}{k}, \tfrac{m}{2}, \tfrac{m}{2}-1\right) = \cdots =$$
$$= (k-1)^{h-1}\cdot E_{\text{AO}_{m,n}}\left(\tfrac{n}{k^{h-1}}, \tfrac{m}{2^{h-1}}, \tfrac{m}{2^{h-1}}-1\right) = (k-1)^{h-1}\cdot E_{\text{AO}_{m,n}}(k,2,1)$$

Thus $E_{\text{AO}_{m,n}}(n,m,m-1) = (k-1)^h$. □

Finally, we note that since $E_{\text{AO}_{m,n}}(n,m,m-1) = n - \log m \cdot o(n)$, the effectiveness of the algorithm comes reasonably close, asymptotically in n, to the corresponding lower bound of $n-f$.

4.3 Open Problems

The multiprocess algorithm $\text{AO}_{m,n}$ scales well when it comes to work complexity, but does not scale well in terms of effectiveness when the number of processes increases. The algorithm applies two techniques. The first one is the collision avoidance strategy, and the second on is having jobs that can be performed by only specific groups of processes. This jobs provide starting points for processes when they join the execution of the algorithm. This means that algorithm $\text{AO}_{m,n}$ cannot be a solution for the strong at-most-once problem. Could we modify the algorithm, to make it a candidate solution for the strong at-most-once problem? One approach would be to have some join service in the algorithm, that provides starting points, for processes joining late the execution.

It would also be interesting to devise algorithm that focus more on the second technique applied by algorithm $\text{AO}_{m,n}$. We believe that such algorithms could have different characteristics than algorithm $\text{AO}_{m,n}$. Specifically we believe that they may scale better in

terms of effectiveness when the number of processes increases.

5 Near Optimal Algorithm KK_β

In this section we present an asynchronous deterministic algorithm called KK_β, that solves the at-most-once problem. The algorithm is presented and analyzed using the Input Output Automata formalism as presented in the Section 2.

5.1 Algorithm KK_β

We present algorithm KK_β, that solves the at-most-once problem. Parameter $\beta \in \mathbb{N}$ is the termination parameter of the algorithm. Algorithm KK_β is defined for all $\beta \geq m$. If $\beta = m$, algorithm KK_β has optimal up to an additive factor of m effectiveness. Note that although $\beta \geq m$ is not necessary in order to prove the correctness of the algorithm, if $\beta < m$ we cannot guarantee termination of algorithm KK_β.

Shared Variables:

$next = \{next_1, \ldots, next_m\}$, $next_q \in \{0, \ldots, n\}$ initially 0

$done = \{done_{1,1}, \ldots, done_{m,n}\}$, $done_{q,i} \in \{0, \ldots, n\}$ initially 0

Signature:

Input:	Internal:	Internal *Read:*	Internal *Write:*
stop_p, $p \in \mathcal{P}$	$\mathsf{compNext}_p$, $p \in \mathcal{P}$	$\mathsf{gatherTry}_p$, $p \in \mathcal{P}$	$\mathsf{setNext}_p$, $p \in \mathcal{P}$
Output:	check_p, $p \in \mathcal{P}$	$\mathsf{gatherDone}_p$, $p \in \mathcal{P}$	done_p, $p \in \mathcal{P}$
$\mathsf{do}_{p,j}$, $p \in \mathcal{P}$, $j \in \mathcal{J}$			

State:

$\mathrm{STATUS}_p \in \{comp\_next, set\_next, gather\_try, gather\_done, check, do, done, end, stop\}$,

initially $\mathrm{STATUS}_p = comp\_next$

$\mathrm{FREE}_p, \mathrm{DONE}_p, \mathrm{TRY}_p \subseteq \mathcal{J}$, initially $\mathrm{FREE}_p = \mathcal{J}$ and $\mathrm{DONE}_p = \mathrm{TRY}_p = \emptyset$

$\mathrm{POS}_p = \{\mathrm{POS}_p(1), \ldots, \mathrm{POS}_p(m)\}$, where $\mathrm{POS}_p(i) \in \{1, \ldots, n\}$, initially $\mathrm{POS}_p(i) = 1$

$\mathrm{NEXT}_p \in \{1, \ldots, n\}$, initially undefined

$\mathrm{Q}_p \in \{1, \ldots, m\}$, initially 1

$\mathrm{TMP}_p \in \{0, \ldots, n\}$, initially undefined

Figure 3: Algorithm KK_β: Shared Variables, Signature and States

The idea behind the algorithm KK_β (see Fig. 3, 4) is quite intuitive and is based on

an algorithm for renaming processes presented by Attiya *et al.* [6]. Each process p, picks a job i to perform, announces (by writing in shared memory) that it is about to perform the job and then checks if it is safe to perform it (by reading the announcements other processes made in the shared memory, and the jobs other processes announced they have performed). If it is safe to perform the job i, process p will proceed with the $\mathsf{do}_{p,i}$ action and then mark the job completed. If it is not safe to perform i, p will release the job. In either case, p picks a new job to perform. In order to pick a new job, p reads from the shared memory and gathers information on which jobs are safe to perform, by reading the announcements that other processes made in the shared memory about the jobs they are about to perform, and the jobs other processes announced they have already performed. Assuming that those jobs are ordered, p splits the set of "free" jobs in m intervals and picks the first job of the interval with rank equal to p's rank. Note that since the information needed in order to decide whether it is safe to perform a specific job and in order to pick the next job to perform is the same, these steps are combined in the algorithm. In Figure 4, we use function $rank(\mathrm{SET}_1, \mathrm{SET}_2, i)$, that returns the element of set $\mathrm{SET}_1 \setminus \mathrm{SET}_2$ that has rank i. If SET_1 and SET_2 have $\mathrm{O}(n)$ elements and are stored in some tree structure like *red-black tree* or some variant of *B-tree*, the operation $rank(\mathrm{SET}_1, \mathrm{SET}_2, i)$, costs $\mathrm{O}(|\mathrm{SET}_2| \log n)$ assuming that $\mathrm{SET}_2 \subseteq \mathrm{SET}_1$.

We will prove that algorithm KK_β has effectiveness $n - (\beta + m - 2)$. For $\beta = O(m)$ this effectiveness is asymptotically optimal for any $m = o(n)$. Note that by Theorem 3.3 the upper bound on effectiveness of the at-most-once problem is $n - f$, where f is the number of failed processes in the system. Next we present algorithm KK_β in more detail.

Shared Variables. *next* is an array with m elements. In the cell $next_q$ of the array process q announces the job it is about to perform. From the structure of algorithm KK_β, only process q writes in cell $next_q$. On the other hand any process may read cell $next_q$.

done is an $m \times n$ matrix. In line q of the matrix, process q announces the jobs it has performed. Each cell of line q contains the identifier of exactly one job that has been performed by process q. Only process q writes in the cells of line q but any process may read them. Moreover, process q updates line q by adding entries at the end of it.

Transitions of process p:

Input stop_p
Effect:
$\text{STATUS}_p \leftarrow stop$

Internal $\mathsf{compNext}_p$
Precondition:
$\text{STATUS}_p = comp\_next$
Effect:
if $\left|\text{FREE}_p \setminus \text{TRY}_p\right| \geq \beta$ **then**
$\text{TMP}_p \leftarrow \frac{\left|\text{FREE}_p\right| - (m-1)}{m}$
if $\text{TMP}_p \geq 1$ **then**
$\text{TMP}_p \leftarrow \left\lfloor (p-1) \cdot \text{TMP}_p \right\rfloor + 1$
$\text{NEXT}_p \leftarrow rank\left(\text{FREE}_p, \text{TRY}_p, \text{TMP}_p\right)$
else
$\text{NEXT}_p \leftarrow rank\left(\text{FREE}_p, \text{TRY}_p, p\right)$
end
$\text{Q}_p \leftarrow 1$
$\text{TRY}_p \leftarrow \emptyset$
$\text{STATUS}_p \leftarrow set\_next$
else
$\text{STATUS}_p \leftarrow end$
end

Internal Write $\mathsf{setNext}_p$
Precondition:
$\text{STATUS}_p = set\_next$
Effect:
$next_p \leftarrow \text{NEXT}_p$
$\text{STATUS}_p \leftarrow gather\_try$

Internal Read $\mathsf{gatherTry}_p$
Precondition:
$\text{STATUS}_p = gather\_try$
Effect:
if $\text{Q}_p \neq p$ **then**
$\text{TMP}_p \leftarrow next_{\text{Q}_p}$
if $\text{TMP}_p > 0$ **then**
$\text{TRY}_p \leftarrow \text{TRY}_p \cup \left\{\text{TMP}_p\right\}$
end
end
if $\text{Q}_p + 1 \leq m$ **then**
$\text{Q}_p \leftarrow \text{Q}_p + 1$
else
$\text{Q}_p \leftarrow 1$
$\text{STATUS}_p \leftarrow gather\_done$
end

Internal Read $\mathsf{gatherDone}_p$
Precondition:
$\text{STATUS}_p = gather\_done$
Effect:
if $\text{Q}_p \neq p$ **then**
$\text{TMP}_p \leftarrow done_{\text{Q}_p, \text{POS}_p(\text{Q}_p)}$
if $\text{POS}_p\left(\text{Q}_p\right) \leq n$ AND $\text{TMP}_p > 0$ **then**
$\text{DONE}_p \leftarrow \text{DONE}_p \cup \left\{\text{TMP}_p\right\}$
$\text{FREE}_p \leftarrow \text{FREE}_p \setminus \left\{\text{TMP}_p\right\}$
$\text{POS}_p\left(\text{Q}_p\right) = \text{POS}_p\left(\text{Q}_p\right) + 1$
else $\text{Q}_p \leftarrow \text{Q}_p + 1$
end
else $\text{Q}_p \leftarrow \text{Q}_p + 1$
end
if $\text{Q}_p > m$ **then**
$\text{Q}_p \leftarrow 1$
$\text{STATUS}_p \leftarrow check$
end

Internal check_p
Precondition:
$\text{STATUS}_p = check$
Effect:
if $\text{NEXT}_p \notin \text{TRY}_p$ AND $\text{NEXT}_p \notin \text{DONE}_p$
then $\text{STATUS}_p \leftarrow do$
else
$\text{STATUS}_p \leftarrow comp\_next$
end

Output $\mathsf{do}_{p,j}$
Precondition:
$\text{STATUS}_p = do$
$\text{NEXT}_p = j$
Effect:
$\text{STATUS}_p \leftarrow done$

Internal Write done_p
Precondition:
$\text{STATUS}_p = done$
Effect:
$done_{p, \text{POS}_p(p)} \leftarrow \text{NEXT}_p$
$\text{DONE}_p \leftarrow \text{DONE}_p \cup \left\{\text{NEXT}_p\right\}$
$\text{FREE}_p \leftarrow \text{FREE}_p \setminus \left\{\text{NEXT}_p\right\}$
$\text{POS}_p\left(p\right) \leftarrow \text{POS}_p\left(p\right) + 1$
$\text{STATUS}_p \leftarrow comp\_next$

Figure 4: Algorithm KK_β: Transitions

Internal Variables of process p. The variable STATUS_p records the status of process p and defines its next action as follows: $\text{STATUS}_p = comp\_next$ - process p is ready to compute the next job to perform (this is the initial status of p), $\text{STATUS}_p = set\_next$ - p computed the next job to perform and is ready to announce it by writing in the shared memory, $\text{STATUS}_p = gather\_try$ - p reads the array *next* in shared memory in order to compute the TRY_p set, $\text{STATUS}_p = gather\_done$ - p reads the matrix *done* in shared memory in order to update the DONE_p and FREE_p sets, $\text{STATUS}_p = check$ - p has to check whether it is safe to perform its current job, $\text{STATUS}_p = do$ - p can safely perform its current job, $\text{STATUS}_p = done$ - p performed its current job and needs to update the shared memory, $\text{STATUS}_p = end$ - p terminated, $\text{STATUS}_p = stop$ - p crashed.

$\text{FREE}_p, \text{DONE}_p, \text{TRY}_p \subseteq \mathcal{J}$ are three sets that are used by process p in order to compute the next job to perform and whether it is safe to perform it. We use some tree structure like *red-black tree* or some variant of *B-tree* [7, 22] for the sets FREE_p, DONE_p and TRY_p, in order to be able to add, remove and search elements in them with $O(\log n)$ work. FREE_p, is initially set to $\mathcal{J}$ and contains an estimate of the jobs that are still available. DONE_p is initially empty and contains an estimate of the jobs that have been performed. No job is removed from DONE_p or added to FREE_p during the execution of algorithm KK_β. TRY_p is initially empty and contains an estimate of the jobs that other processes are about to perform. It holds that $\left|\text{TRY}_p\right| < m$, since there are $m-1$ processes apart from process p that may be attempting to perform a job.

POS_p is an array of m elements. Position $\text{POS}_p(q)$ of the array contains a pointer in the line q of the shared matrix *done*. $\text{POS}_p(q)$ is the element of line q that process p will read from. In the special case where $q = p$, $\text{POS}_p(p)$ is the element of line p that process p will write into after performing a new job. The elements of the shared matrix *done* are read when process p is updating the DONE_p set.

NEXT_p contains the job process p is attempting to perform.

TMP_p is a temporary storage for values read from the shared memory.

$\text{Q}_p \in \{1, \ldots, m\}$ is used as indexing for looping through process identifiers.

Actions of process p. We visit them one by one below.

$\mathsf{compNext}_p$: Process p computes the set $\text{FREE}_p \setminus \text{TRY}_p$ and if it has more or equal elements to β, were β is the termination parameter of the algorithm, process p computes its next candidate job, by splitting the $\text{FREE}_p \setminus \text{TRY}_p$ set in m parts and picking the first element of the p-th part. In order to do that it uses the function $rank(\text{SET}_1, \text{SET}_2, i)$, which returns the element of set $\text{SET}_1 \setminus \text{SET}_2$ with rank i. Finally process p sets the TRY_p set to the empty set, the Q_p internal variable to 1 and its status to *set_next* in order to update the shared memory with its new candidate job. If the $\text{FREE}_p \setminus \text{TRY}_p$ set has less than β elements process p terminates.

$\mathsf{setNext}_p$: Process p announces its new candidate job by writing the contents of its NEXT_p internal variable in the p-th position of the *next* array. Remember that the *next* array is stored in shared memory. Process p changes its status to *gather_try*, in order to start collecting the TRY_p set from the *next* array.

$\mathsf{gatherTry}_p$: With this action process p implements a loop, which reads from the shared memory all the positions of the array *next* and updates the TRY_p set. In each execution of the action, process p checks if Q_p is equal to p. If it is not equal, p reads the Q_p-th position of the array *next*, checks if the value read is greater than 0 and if it is, adds the value it read in the TRY_p set. If Q_p is equal to p, p just skips the step described above. Then p checks if the value of $\text{Q}_p + 1$ is less than $m + 1$. If it is, then p increases Q_p by 1 and leaves its status *gather_try*, otherwise p has finished updating the TRY_p set and thus sets Q_p to 1 and changes its status to *gather_done*, in order to update the DONE_p and FREE_p sets from the contents of the *done* matrix.

$\mathsf{gatherDone}_p$: With this action process p implements a loop, which updates the DONE_p and FREE_p sets with values read from the matrix *done*, which is stored in shared memory. In each execution of the action, process p checks if Q_p is equal to p. If it is not equal, p uses the internal variable $\text{POS}_p\left(\text{Q}_p\right)$, in order to read fresh values from the line Q_p of the *done* matrix. In detail, p reads the shared variable $done_{\text{Q}_p,\text{POS}_p\left(\text{Q}_p\right)}$, checks if $\text{POS}_p\left(\text{Q}_p\right)$ is less than $n+1$ and if the value read is greater than 0. If both conditions hold, p adds the value read at the DONE_p set, removes the value read from the FREE_p set and increases $\text{POS}_p\left(\text{Q}_p\right)$ by one. Otherwise, it means that either process Q_p has terminated (by performing all the n jobs) or the line Q_p does not contain any new completed jobs. In either case p increases the value of Q_p by 1. The value of Q_p is increased by 1 also if Q_p was equal to p. Finally p checks whether Q_p is greater than m; if it is, p has completed the loop and thus changes its status to *check*.

check_p: Process p checks if it is safe to perform its current job. This is done by checking if NEXT_p belongs to the set TRY_p or to the set DONE_p. If it does not, then it is safe to perform the job NEXT_p and p changes its status to *do*. Otherwise it is not safe, and thus p changes its status to *comp_next*, in order to find a new job that may be safe to perform.

$\mathsf{do}_{p,j}$: Process p performs job j. Note that $\text{NEXT}_p = j$ is part of the preconditions for the action to be enabled in a state. Then p changes its status to *done*.

done_p: Process p writes in the $done_{p,\text{POS}_p(p)}$ position of the shared memory the value of NEXT_p, letting other processes know that it performed job NEXT_p. Also p adds NEXT_p to its DONE_p set, removes NEXT_p from its FREE_p set, increases $\text{POS}_p(p)$ by 1 and changes its status to *comp_next*.

stop_p: Process p crashes by setting its status to *stop*.

5.2 Correctness and Effectiveness Analysis

We begin the analysis of algorithm KK_β, by showing in Lemma 5.1 that KK_β solves the at-most-once problem. That is, there exists no execution of KK_β in which 2 distinct actions $\mathsf{do}_{p,i}$ and $\mathsf{do}_{q,i}$ appear for some $i \in \mathcal{J}$ and $p,q \in \mathcal{P}$. We continue the analysis by showing in Theorem 5.4 that algorithm KK_β has effectiveness $E_{\text{KK}_\beta}(n,m,f) = n - (\beta + m - 2)$. This is done in two steps. First in Lemma 5.2, we show that algorithm KK_β cannot terminate its execution if less than $n - (\beta + m - 1)$ jobs are performed. The effectiveness analysis is completed by showing in Lemma 5.3, that the algorithm is wait-free (it has no infinite fair executions). In Theorem 5.4 we combine the two lemmas in order to show that the effectiveness of algorithm KK_β is greater that or equal to $n - (\beta + m - 2)$. Moreover, we show the existence of an adversarial strategy, that results in a terminating execution where $n - (\beta + m - 2)$ jobs are completed, showing that the bound is tight.

In the analysis that follows, for a state s and a process p we denote by $s.\text{FREE}_p$, $s.\text{DONE}_p$, $s.\text{TRY}_p$, the values of the internal variables FREE, DONE and TRY of process p in state s. Moreover with $s.next$, and $s.done$ we denote the contents of the array $next$ and the matrix $done$ in state s. Remember that $next$ and $done$, are stored in shared memory.

Lemma 5.1. *There exists no execution α of algorithm KK_β, such that $\exists i \in \mathcal{J}$ and $\exists p,q \in \mathcal{P}$ for which $\mathsf{do}_{p,i}, \mathsf{do}_{q,i} \in \alpha$.*

Proof. Let us for the sake of contradiction assume that there exists an execution $\alpha \in execs(\text{KK}_\beta)$ and $i \in \mathcal{J}$ and $p,q \in \mathcal{P}$ such that $\mathsf{do}_{p,i}, \mathsf{do}_{q,i} \in \alpha$. We examine two cases.

Case 1 $p = q$**:** Let states $s_1, s_1', s_2, s_2' \in \alpha$, such that the transitions $\left(s_1, \mathsf{do}_{p,i}, s_1'\right)$, $\left(s_2, \mathsf{do}_{p,i}, s_2'\right) \in \alpha$ and without loss of generality assume $s_1' \leq s_2$ in α. From Figure 4

we have that $s_1^{'}.\text{NEXT}_p = i$, $s_1^{'}.\text{STATUS}_p = done$ and $s_2.\text{NEXT}_p = i$, $s_2.\text{STATUS}_p = do$. From algorithm KK_β, state s_2 must be preceded by transition $\left(s_3, \mathsf{check}_p, s_3^{'}\right)$, such that $s_3.\text{NEXT}_p = i$ and $s_3^{'}.\text{NEXT}_p = i$, $s_3^{'}.\text{STATUS}_p = do$, where $s_1^{'}$ precedes s_3 in α. Finally s_3 must be preceded in α by transition $\left(s_4, \mathsf{done}_p, s_4^{'}\right)$, where $s_1^{'}$ precedes s_4, such that $s_4.\text{NEXT}_p = i$ and $i \in s_4^{'}.\text{DONE}_p$. Since $s_4^{'}$ precedes s_3 and during the execution of KK_β no elements are removed from DONE_p, we have that $i \in s_3.\text{DONE}_p$. This is a contradiction, since the transition $(\{\text{NEXT}_p = i, i \in \text{DONE}_p\}, \mathsf{check}_p, \{\text{NEXT}_p = i, \text{STATUS}_p = do\})$ $\notin trans(\text{KK}_\beta)$.

Case 2 $p \neq q$**:** Given transition $\left(s_1, \mathsf{do}_{p,i}, s_1^{'}\right)$ in execution α, we deduce from Fig. 4 that there exist in α transitions $\left(s_2, \mathsf{setNext}_p, s_2^{'}\right)$, $\left(s_3, \mathsf{gatherTry}_p, s_3^{'}\right)$, $\left(s_4, \mathsf{check}_p, s_4^{'}\right)$, where $s_2^{'}.next_p = s_2^{'}.\text{NEXT}_p = i$, $s_3.next_p = s_3.\text{NEXT}_p = i, s_3.\text{Q}_p = q$, $s_4.\text{NEXT}_p = i$, $s_4^{'}.\text{NEXT}_p = i$, $s_4^{'}.\text{STATUS}_p = do$, such that $s_2 < s_3 < s_4 < s_1$ and there exists no action $\pi = \mathsf{compNext}_p$ in execution α, such that $s_2 < \pi < s_1^{'}$.

Similarly for transition $\left(t_1, \mathsf{do}_{q,i}, t_1^{'}\right)$ there exist in execution α transitions $\left(t_2, \mathsf{setNext}_q, t_2^{'}\right)$, $\left(t_3, \mathsf{gatherTry}_q, t_3^{'}\right)$, $\left(t_4, \mathsf{check}_q, t_4^{'}\right)$, where $t_2^{'}.next_q = t_2^{'}.\text{NEXT}_q = i$, $t_3.next_q = t_3.\text{NEXT}_q = i, t_3.\text{Q}_q = p$, $t_4.\text{NEXT}_q = i$, $t_4^{'}.\text{NEXT}_q = i$, $t_4^{'}.\text{STATUS}_q = do$, such that $t_2 < t_3 < t_4 < t_1$ and there exists no action $\pi' = \mathsf{compNext}_q$ in execution α, such that $t_2 < \pi < t_1^{'}$.

Either state $s_2 < t_3$ or $t_3 < s_2$ which implies $t_2 < s_3$. We will show that if $s_2 < t_3$ then $\mathsf{do}_{q,i}$ cannot take place, leading to a contradiction. The case where $t_2 < s_3$ is symmetric and will be omitted.

Let us assume that s_2 precedes t_3. We have two cases, either $t_3.next_p = i$ or $t_3.next_p \neq i$. In the first case $i \in t_3^{'}.\text{TRY}_q$. The only action in which entries are removed from the TRY_q set, is action $\mathsf{compNext}_q$, where the TRY_q set is reset to $\emptyset$. Thus $i \in t_4.\text{TRY}_q$, since $\nexists\, \pi' = \mathsf{compNext}_q \in \alpha$, such that $t_2 < \pi' < t_1$. This is a contradiction since $\left(t_4, \mathsf{check}_q, t_4^{'}\right) \notin$

$trans(\text{KK}_\beta)$, if $i \in t_4.\text{TRY}_q$, $t_4.\text{NEXT}_q = i$ and $t_4^{'}.\text{STATUS}_q = do$.

If $t_3.next_p \neq i$, since $\left(s_2, \text{setNext}_p, s_2^{'}\right) \in \alpha$ and $s_2^{'} < t_3$ there exists action $\pi_1 = \text{setNext}_p \in \alpha$, such that $s_2^{'} < \pi_1 < t_3$. Moreover, there exists action $\pi_2 = \text{compNext}_p$ in α, such that $s_2^{'} < \pi_2 < \pi_1$. Since $\nexists\, \pi = \text{compNext}_p \in \alpha$, such that $s_2 < \pi < s_1^{'}$, it holds that $s_1^{'} < \pi_2 < \pi_1 < t_3$. Furthermore, from Fig. 4 there exists transition $\left(s_5, \text{done}_p, s_5^{'}\right)$ in α and $j \in \{1, \ldots, n\}$, such that $s_5.\text{POS}_p(p) = j$, $s_5.done_{p,j} = 0$, $s_5.\text{NEXT}_p = i$, $s_5^{'}.done_{p,j} = i$ and $s_1^{'} < s_5^{'} < \pi_2 < t_3$. It must be the case that $i \notin t_2.DONE_q$, since $t_2.\text{NEXT}_q = i$. From that and from Fig. 4 we have that there exists transition $\left(t_6, \text{gatherDone}_q, t_6^{'}\right)$ in α, such that $t_6.\text{Q}_q = p$, $t_6.\text{POS}_q(p) = j$ and $t_3 < t_6 < t_4$. Since $s_5^{'} < t_3$ and $done_{p,j}$ from algorithm KK_β cannot be changed again in execution α, we have that $t_6.done_{p,j} = i$ and as a result $i \in t_6^{'}.\text{DONE}_q$. Moreover, during the execution of algorithm KK_β, entries in set DONE_q are only added and never removed, thus we have that $i \in t_4.DONE_q$. This is a contradiction since $\left(t_4, \text{check}_q, t_4^{'}\right) \notin trans(\text{KK}_\beta)$, if $i \in t_4.\text{DONE}_q$, $t_4.\text{NEXT}_q = i$ and $t_4^{'}.\text{STATUS}_q = do$. This completes the proof.

□

Next we examine the effectiveness of the algorithm. First we show that algorithm KK_β cannot terminate its execution if less than $n - (\beta + m - 1)$ jobs are performed.

Lemma 5.2. *For any $\beta \geq m$, $f \leq m-1$ and for any finite execution $\alpha \in execs\left(\text{KK}_\beta\right)$ with $Do(\alpha) \leq n - (\beta + m - 1)$, there exists a (non-empty) execution fragment α' such that $\alpha \cdot \alpha' \in execs\left(\text{KK}_\beta\right)$.*

Proof. From the algorithm KK_β, we have that for any process p and any state $s \in \alpha$, $\left|s.\text{FREE}_p\right| \geq n - Do(\alpha)$ and $\left|s.\text{TRY}_p\right| \leq m - 1$. The first inequality holds since the $s.\text{FREE}_p$ set is estimated by p by examining the *done* matrix which is stored in shared memory. From algorithm KK_β, a job j is only inserted in line q of the matrix *done*, if a

$\mathsf{do}_{q,j}$ action has already been performed by process q. The second inequality is obvious. Thus we have that $\forall p \in \mathcal{P}$ and $\forall s \in \alpha$, $\left|s.\text{FREE}_p \setminus s.\text{TRY}_p\right| \geq n - (Do(\alpha) + m - 1)$. If $Do(\alpha) \leq n - (\beta + m - 1)$, $\forall p \in \mathcal{P}$ and $\forall s \in \alpha$ we have that $\left|s.\text{FREE}_p \setminus s.\text{TRY}_p\right| \geq \beta$. Since there can be $f \leq m-1$ failed processes in our system, at the final state s' of execution α there exists at least one process $p \in \mathcal{P}$ that has not failed. This process has not terminated, since from Fig. 4 a process p can only terminate if in the enabling state s of action $\mathsf{compNext}_p$, $\left|s.\text{FREE}_p \setminus s.\text{TRY}_p\right| < \beta$. This process can continue executing steps and thus there exists a (non-empty) execution fragment α' such that $\alpha \cdot \alpha' \in execs\left(\text{KK}_\beta\right)$.

□

Since no finite execution of algorithm KK_β can terminate if less than $n - (\beta + m - 1)$ jobs are performed, Lemma 5.2 implies that if the algorithm KK_β has effectiveness less than or equal to $n - (\beta + m - 1)$, there must exist some infinite fair execution α with $Do(\alpha) \leq n - (\beta + m - 1)$. Next we prove that algorithm KK_β is wait-free (it has no infinite fair executions).

Lemma 5.3. *For any $\beta \geq m$, $f \leq m-1$ there exists no infinite fair execution $\alpha \in execs(\text{KK}_\beta)$.*

Proof. We will prove this by contradiction. Let $\beta \geq m$ and $\alpha \in execs(\text{KK}_\beta)$ an infinite fair execution with $f \leq m-1$ failures, and let $Do(\alpha)$ be the jobs executed by execution α according to Definition 2.1. Since $\alpha \in execs(\text{KK}_\beta)$ and from Lemma 5.1 KK_β solves the at-most-once problem, $Do(\alpha)$ is finite. Clearly there exists at least one process in execution α that has not crashed and does not terminate (some process must take steps in α in order for it to be infinite). Since $Do(\alpha)$ and f are finite, there exists a state s_0 in α such that after s_0 no process crashes, no process terminates, no do action takes place in α and no process adds new entries in the *done* matrix in shared memory. The later holds since the execution is infinite and fair, the $Do(\alpha)$ is also finite, consequently any

non failed process q that has not terminated will eventually update the q line of the *done* matrix to be in agreement with the $\mathsf{do}_{q,*}$ actions it has performed. Moreover any process q that has terminated, has already updated the q line of *done* matrix with the latest do action it performed, before it terminated, since in order to terminate it must have reached a compNext action that has set its status to *end*.

We define the following sets of processes and jobs according to state s_0. $\mathcal{J}_\alpha$ are jobs that have been performed in α according to Definition 2.1. $\mathcal{P}_\alpha$ are processes that do not crash and do not terminate in α. By the way we defined state s_0 only processes in $\mathcal{P}_\alpha$ take steps in α after state s_0. $\text{STUCK}_\alpha = \{i \in \mathcal{J} \setminus \mathcal{J}_\alpha | \exists \text{ failed process } p : s_0.next_p = i\}$, i.e., STUCK_α expresses the set of jobs that are held by failed processes. $\text{DONE}_\alpha = \{i \in \mathcal{J}_\alpha | \exists p \in \mathcal{P} \text{ and } j \in \{1,\ldots,n\} : s_0.done_p(j) = i\}$, i.e., DONE_α expresses the set of jobs that have been performed before state s_0 and the processes that performed them managed to update the shared memory. Finally we define $\text{POOL}_\alpha = \mathcal{J} \setminus (\mathcal{J}_\alpha \cup \text{STUCK}_\alpha)$. After state s_0, all processes in $\mathcal{P}_\alpha$ will keep executing. This means that whenever a process $p \in \mathcal{P}_\alpha$ takes action $\mathsf{compNext}_p$ in α, the first if statement is true. Specifically it holds that for $\forall p \in \mathcal{P}_\alpha$ and for all the enabling states $s \geq s_0$ of actions $\mathsf{compNext}_p$ in α, $\left|\text{FREE}_p \setminus \text{TRY}_p\right| \geq \beta$.

From Figure 4, we have that for any $p \in \mathcal{P}_\alpha$, $\exists\, s_p \in \alpha$ such that $s_p > s_0$ and for all states $s \geq s_p$, $s.\text{DONE}_p = \text{DONE}_\alpha$, $s.\text{FREE}_p = \mathcal{J} \setminus \text{DONE}_\alpha$ and $s.\text{FREE}_p \setminus s.\text{TRY}_p \subseteq \text{POOL}_\alpha$. Let $s'_0 = \max_{p \in \mathcal{P}_\alpha}[s_p]$. From the above we have: $|\mathcal{J} \setminus \text{DONE}_\alpha| \geq \beta \geq m$ and $|\text{POOL}_\alpha| \geq \beta \geq m$, since $\forall s' \geq s'_0$ we have that $s'.\text{FREE}_p = \mathcal{J} \setminus \text{DONE}_\alpha$ and $s'.\text{FREE}_p \setminus s'.\text{TRY}_p \subseteq \text{POOL}_\alpha$ and $\forall p \in \mathcal{P}_\alpha$ and for all the enabling states $s \geq s'_0$ of actions $\mathsf{compNext}_p$ in α, we have that $\left|\text{FREE}_p \setminus \text{TRY}_p\right| \geq \beta$.

Let p_0 be the process with the smallest process identifier in $\mathcal{P}_\alpha$. We examine 2 cases according to the size of $\mathcal{J} \setminus \text{DONE}_\alpha$.

Case A $|\mathcal{J} \setminus \text{DONE}_\alpha| \geq 2m-1$: Let $x_0 \in \text{POOL}_\alpha$ be the job such that $[x_0]_{\text{POOL}_\alpha} = \left\lfloor (p_0-1) \cdot \frac{|\mathcal{J} \setminus \text{DONE}_\alpha| - (m-1)}{m} \right\rfloor + 1$. Such x_0 exists since $\forall p \in \mathcal{P}_\alpha$ and $\forall s \geq s'_0$ it holds $s.\text{FREE}_p \setminus s.\text{TRY}_p \subseteq \text{POOL}_\alpha$, $s.\text{FREE}_p = \mathcal{J} \setminus \text{DONE}_\alpha$ from which we have that $|\text{POOL}_\alpha| \geq |\mathcal{J} \setminus \text{DONE}_\alpha| - \left|s.\text{TRY}_p\right| \geq |\mathcal{J} \setminus \text{DONE}_\alpha| - (m-1) \geq m$.

It follows that any $p \in \mathcal{P}_\alpha$ that executes action $\mathsf{compNext}_p$ after state s'_0, will have its NEXT_p variable pointing in a job x with $[x]_{\text{POOL}_\alpha} \geq \left\lfloor (p-1) \cdot \frac{|\mathcal{J} \setminus \text{DONE}_\alpha| - (m-1)}{m} \right\rfloor + 1$. Thus $\forall p \in \mathcal{P}_\alpha$, $\exists\ s'_p \geq s'_0$ in α such that $\forall$ states $s \geq s'_p$, $[s.next_p]_{\text{POOL}_\alpha} \geq \left\lfloor (p-1) \cdot \frac{|\mathcal{J} \setminus \text{DONE}_\alpha| - (m-1)}{m} \right\rfloor + 1$. Let $s''_0 = \max_{p \in \mathcal{P}_\alpha}[s'_p]$, we have 2 cases for p_0:

Case A.1) After s''_0, process p_0 executes action $\mathsf{compNext}_{p_0}$ and the transition leads to state $s_1 > s''_0$ such that $s_1.\text{NEXT}_{p_0} = x_0$. Since $[x_0]_{\text{POOL}_\alpha} = \left\lfloor (p_0-1) \cdot \frac{|\mathcal{J} \setminus \text{DONE}_\alpha| - (m-1)}{m} \right\rfloor + 1$ and $p_0 = \min_{p \in \mathcal{P}_\alpha}[p]$, from the previous discussion we have that $\forall s \geq s_1$ and $\forall p \in \mathcal{P} \setminus \{p_0\}$, $s.next_p \neq x_0$. Thus when p_0 executes action check_p of Fig. 4 for the first time after state s_1, the condition will be true, so in some subsequent transition p_0 will have to execute action do_{p_0,x_0}, performing job x_0, which is a contradiction, since after state s_0 no jobs are executed.

Case A.2) After s''_0, process p_0 executes action $\mathsf{compNext}_{p_0}$ and the transition leads in state $s_1 > s''_0$ such that $s_1.\text{NEXT}_{p_0} > x_0$. Since $p_0 = \min_{p \in \mathcal{P}_\alpha}[p]$, it holds that $\forall x \in \text{POOL}_\alpha$ such that $[x]_{\text{POOL}_\alpha} \leq \left\lfloor (p_0-1) \cdot \frac{|\mathcal{J} \setminus \text{DONE}_\alpha| - (m-1)}{m} \right\rfloor + 1$, $\nexists p \in \mathcal{P}$ such that $s_1.next_p = x$. Let the transition $\left(s_2, \mathsf{compNext}_{p_0}, s'_2\right) \in \alpha$, where $s_2 > s_1$, be the first time that action $\mathsf{compNext}_{p_0}$ is executed after state s_1. We have that $\forall x \in \text{POOL}_\alpha$ such that $[x]_{\text{POOL}_\alpha} \leq \left\lfloor (p_0-1) \cdot \frac{|\mathcal{J} \setminus \text{DONE}_\alpha| - (m-1)}{m} \right\rfloor + 1$, $x \notin s_2.\text{DONE}_{p_0} \cup s_2.\text{TRY}_{p_0}$, since from the discussion above we have that $\forall s \geq s_1$ and $\forall p \in \mathcal{P}_\alpha \setminus \{p_0\}$, $[s.next_p]_{\text{POOL}_\alpha} \geq \left\lfloor (p-1) \cdot \frac{|\mathcal{J} \setminus \text{DONE}_\alpha| - (m-1)}{m} \right\rfloor + 1$. Thus $[x_0]_{s_2.\text{FREE}_{p_0} \setminus s_2.\text{TRY}_{p_0}} = [x_0]_{\text{POOL}_\alpha} = \left\lfloor (p_0-1) \cdot \frac{|\mathcal{J} \setminus \text{DONE}_\alpha| - (m-1)}{m} \right\rfloor + 1$. As a result, $s'_2.\text{NEXT}_{p_0} = x_0$. With similar arguments like in case A.1, we can see that job x_0 will be performed by process p_0,

which is a contradiction, since after state s_0 no jobs are executed.

Case B $|\mathcal{J} \setminus \text{DONE}_\alpha| < 2m-1$: Let $x_0 \in \text{POOL}_\alpha$ be the job such that $[x_0]_{\text{POOL}_\alpha} = p_0$. Such x_0 exists since $\beta \geq m$ and $\text{POOL}_\alpha \geq \beta$. It follows that any $p \in \mathcal{P}_\alpha$ that executes action $\mathsf{compNext}_p$ after state s'_0, will have its NEXT_p variable pointing in a job x with $[x]_{\text{POOL}_\alpha} \geq p$. Thus $\forall p \in \mathcal{P}_\alpha$, $\exists\, s'_p \geq s'_0$ in α such that $\forall$ states $s \geq s'_p$, $[s.next_p]_{\text{POOL}_\alpha} \geq p$. Let $s''_0 = \max_{p \in \mathcal{P}_\alpha}[s'_p]$, we have 2 cases for p_0:

Case B.1) After s''_0, process p_0 executes action $\mathsf{compNext}_{p_0}$ and the transition leads in state $s_1 > s''_0$ such that $s_1.\text{NEXT}_{p_0} = x_0$. Since $[x_0]_{\text{POOL}_\alpha} = p_0$ and $p_0 = \min_{p \in \mathcal{P}_\alpha}[p]$, from the previous discussion we have that $\forall s \geq s_1$ and $\forall p \in \mathcal{P} \setminus \{p_0\}$, $s.next_p \neq x_0$. Thus when p_0 executes action check_p of Fig. 4 for the first time after state s_1, the condition will be true, so in some subsequent transition p_0 will have to execute action do_{p_0,x_0}, performing job x_0, which is a contradiction, since after state s_0 no jobs are executed.

Case B.2) After s''_0, process p_0 executes action $\mathsf{compNext}_{p_0}$ and the transition leads in state $s_1 > s''_0$ such that $s_1.\text{NEXT}_{p_0} > x_0$. Since $p_0 = \min_{p \in \mathcal{P}_\alpha}[p]$, it holds that $\forall x \in \text{POOL}_\alpha$ such that $[x]_{\text{POOL}_\alpha} \leq p_0$, $\nexists p \in \mathcal{P}$ such that $s_1.next_p = x$. Let the transition $\left(s_2, \mathsf{compNext}_{p_0}, s'_2\right) \in \alpha$, where $s_2 > s_1$, be the first time that action $\mathsf{compNext}_{p_0}$ is executed after state s_1. We have that $\forall x \in \text{POOL}_\alpha$ such that $[x]_{\text{POOL}_\alpha} \leq p_0$, $x \notin s_2.\text{DONE}_{p_0} \cup s_2.\text{TRY}_{p_0}$, since from the discussion above we have that $\forall s \geq s_1$ and $\forall p \in \mathcal{P}_\alpha \setminus \{p_0\}$, $[s.next_p]_{\text{POOL}_\alpha} \geq p$. Thus $[x_0]_{s_2.\text{FREE}_{p_0} \setminus s_2.\text{TRY}_{p_0}} = [x_0]_{\text{POOL}_\alpha} = p_0$. As a result, $s'_2.\text{NEXT}_{p_0} = x_0$. With similar arguments like in case B.1, we can see that job x_0 will be performed by process p_0, which is a contradiction, since after state s_0 no jobs are executed.

□

We combine the last two lemmas in order to show the main result on the effectiveness of algorithm KK_β.

Theorem 5.4. *For any $\beta \geq m$, $f \leq m-1$ algorithm KK_β has effectiveness $E_{\mathrm{KK}_\beta}(n,m,f) = n-(\beta+m-2)$.*

Proof. From Lemma 5.2 we have that any finite execution $\alpha \in \mathit{execs}\left(\mathrm{KK}_\beta\right)$ with $Do(\alpha) \leq n-(\beta+m-1)$ can be extended, essentially proving that in such executions no process has terminated. Moreover from Lemma 5.3 we have that KK_β is wait free, and thus there exists no infinite fair execution $\alpha \in \mathit{execs}\left(\mathrm{KK}_\beta\right)$, such that $Do(\alpha) \leq n-(\beta+m-1)$. Since finite fair executions are executions where all non-failed processes have terminated, from the above we have that $E_{\mathrm{KK}_\beta}(n,m,f) \geq n-(\beta+m-2)$.

If all processes but the process with id m fail in an execution α in such a way that $\mathcal{J}_\alpha \cap \mathrm{STUCK}_\alpha = \emptyset$ and $|\mathrm{STUCK}_\alpha| = m-1$ (where STUCK_α is defined as in the proof of Lemma 5.3), it is easy to see that there exists an adversarial strategy, such that when process m terminates, $\beta+m-2$ jobs have not been performed . Such an execution will be a finite fair execution where $n-(\beta+m-2)$ jobs are performed. Thus we have that $E_{\mathrm{KK}_\beta}(n,m,f) = n-(\beta+m-2)$. □

5.3 Work Complexity Analysis

In this section we are going to prove that for $\beta \geq 3m^2$ algorithm KK_β has work complexity $\mathrm{O}(nm\log n\log m)$.

The main idea of the proof, is to demonstrate that under the assumption $\beta \geq 3m^2$, process *collisions* on a job cannot accrue without making progress in the algorithm. In order to prove that, we first demonstrate in Lemma 5.5 that if two different processes p,q set their $\mathrm{NEXT}_p, \mathrm{NEXT}_q$ internal variables to the same job i in some compNext actions, then the DONE_p and DONE_q sets of the processes, have at least $|q-p|\cdot m$ different elements, given that $\beta \geq 3m^2$. Next we prove in Lemma 5.8 that if two processes p,q collide three consecutive times, while trying to perform some jobs, the size of the set $\mathrm{DONE}_p \cup \mathrm{DONE}_q$

that processes p and q know will increase by at least $|q-p| \cdot m$ elements. This essentially tells us that every three collisions between the same two processes a significant number of jobs has been performed, and thus enough progress has been made. In order to prove the above statement, we formally define what we mean by collision in Definition 5.2, and tie such a collision with some specific state, the state the collision is detected, so that we have a fixed "point of reference" in the execution; and show that the order collisions are detected in an execution, is consistent with the order the involved processes attempt to perform the respective jobs in Lemmas 5.6, 5.7. Finally we use Lemma 5.8, in order to prove in Lemma 5.9, that a process p cannot collide with a process q more than $2\left\lceil \frac{n}{m \cdot |q-p|} \right\rceil$ times in any execution. This is proven by contradiction, showing that if process p collides with process q more than $2\left\lceil \frac{n}{m \cdot |q-p|} \right\rceil$ times, there exist states for which the set $|\text{DONE}_p \cup \text{DONE}_q|$ has more than n elements which is impossible. Lemma 5.9 is used in order to prove the main result on the work complexity of algorithm KK_β for $\beta \geq 3m^2$, Theorem 5.10. We obtain Theorem 5.10 by counting the total number of collisions that can happen and the cost of each collision.

We start by defining the notion of *immediate predecessor* transition for a state s in an execution α. The immediate predecessor is the last transition of a specific action type that precedes state s in the execution. This is particularly useful in uniquely identifying the transition with action $\mathsf{compNext}_p$ in an execution, that last set a NEXT_p internal variable to a specific value, given a state s of interest.

Definition 5.1. *We say that transition* $\left(s_1, \pi_1, s_1^{'}\right)$ *is an* ***immediate predecessor*** *of state* s_2 *in an execution* $\alpha \in execs(\text{KK}_\beta)$ *and we write* $\left(s_1, \pi_1, s_1^{'}\right) \mapsto s_2$*, if* $s_1^{'} < s_2$ *and in the execution fragment* α' *that begins with state* $s_1^{'}$ *and ends with state* s_2*, there exists no action* $\pi_3 = \pi_1$.

Next we define what a collision between two processes means. We say that process p

collided with process q in job i at state s, if process p attempted to preform job i, but was not able to, because it detected in state s that either process q was trying to perform job i or process q has already performed job i.

Definition 5.2. *In an execution $\alpha \in execs(\mathrm{KK}_\beta)$, we say that process p **collided** with process q in job i at state s, if (i) there exist in α transitions $\left(s_1, \mathsf{compNext}_p, s_1'\right)$, $\left(t_1, \mathsf{compNext}_q, t_1'\right)$ and $\left(s_2, \mathsf{check}_p, s_2'\right)$, where $\left(s_1, \mathsf{compNext}_p, s_1'\right) \mapsto s_2$, $t_1 < s_2$ and $s_1'.\mathrm{NEXT}_p = t_1'.\mathrm{NEXT}_q = s_2.\mathrm{NEXT}_p = i$, $s_1'.\mathrm{STATUS}_p = t_1'.\mathrm{STATUS}_q = set\_next$, $s_2'.\mathrm{STATUS}_p = comp\_next$, (ii) in execution fragment $\alpha' = s_1', \ldots, s_2$ either there exists transition $\left(s, \mathsf{gatherTry}_p, s'\right)$ such that $s.\mathrm{Q}_p = q, s.next_q = i$, or transition $\left(s, \mathsf{gatherDone}_p, s'\right)$ and $j \in \{1, \ldots, n\}$ such that $s.\mathrm{Q}_p = q$, $s.\mathrm{POS}_p(q) = j$, $s.done_{q,j} = i$ and $i \notin s.\mathrm{TRY}_p$.*

Definition 5.3. *In an execution $\alpha \in execs(\mathrm{KK}_\beta)$, we say that processes p, q **collide** in job i at state s, if process p collided with process q or process q collided with process p in job i at state s, according to Definition 5.2.*

Next we show that if two processes p, q decide, with some compNext actions, to perform the same job i, then their DONE sets at the enabling states of those compNext actions, differ in at-least $|q-p| \cdot m$ elements.

Lemma 5.5. *If $\beta \geq 3m^2$ and in an execution $\alpha \in execs(\mathrm{KK}_\beta)$ there exist states s_1, t_1 and processes $p, q \in \mathcal{P}$ with $p < q$ such that $s_1.\mathrm{NEXT}_p = t_1.\mathrm{NEXT}_q = i \in \mathcal{J}$, then there exist transitions $\left(s_2, \mathsf{compNext}_p, s_2'\right) \mapsto s_1$, $\left(t_2, \mathsf{compNext}_q, t_2'\right) \mapsto t_1$, where $s_2'.\mathrm{NEXT}_p = t_2'.\mathrm{NEXT}_q = i$, $s_2'.\mathrm{STATUS}_p = t_2'.\mathrm{STATUS}_q = set\_next$ and $\left|s_2.\mathrm{DONE}_p \cap \overline{t_2.\mathrm{DONE}_q}\right| > (q-p) \cdot m$ or $\left|\overline{s_2.\mathrm{DONE}_p} \cap t_2.\mathrm{DONE}_q\right| > (q-p) \cdot m$.*

Proof. We will prove this by contradiction. From algorithm KK_β there must exist transitions $\left(s_2, \mathsf{compNext}_p, s_2'\right) \mapsto s_1$ and $\left(t_2, \mathsf{compNext}_q, t_2'\right) \mapsto t_1$, where $s_2'.\mathrm{NEXT}_p = i$ and

$t_2^{'}.\text{NEXT}_q = i$, if there exist $s_1, t_1 \in \alpha$ and $p, q \in \mathcal{P}$ with $p < q$ such that $s_1.\text{NEXT}_p = t_1.\text{NEXT}_q = i \in \mathcal{J}$, since those are the transitions that set NEXT_p and NEXT_q to i. In order to get a contradiction we assume that $\left|s_2.\text{DONE}_p \cap \overline{t_2.\text{DONE}_q}\right| \leq (q-p) \cdot m$ and $\left|\overline{s_2.\text{DONE}_p} \cap t_2.\text{DONE}_q\right| \leq (q-p) \cdot m$. We will prove that if this is the case, then $s_2^{'}.\text{NEXT}_p \neq t_2^{'}.\text{NEXT}_q$.

Let $\text{A} = \mathcal{J} \setminus s_2.\text{DONE}_p = s_2.\text{FREE}_p$ and $\text{B} = \mathcal{J} \setminus t_2.\text{DONE}_q = t_2.\text{FREE}_q$, thus from the contradiction assumption we have that: $\left|\overline{\text{A}} \cap \text{B}\right| \leq (q-p) \cdot m$ and $\left|\text{A} \cap \overline{\text{B}}\right| \leq (q-p) \cdot m$.

It could either be that $|\text{A}| < |\text{B}|$ or $|\text{A}| \geq |\text{B}|$.

Case 1 $|\text{A}| < |\text{B}|$**:** From the contradiction assumption we have that $\left|\overline{\text{A}} \cap \text{B}\right| \leq (q-p) \cdot m$. Since $s_2.\text{FREE}_p \setminus s_2.\text{TRY}_p$ can have up to $m-1$ fewer elements than A – the elements of set $s_2.\text{TRY}_p$ – and it can be the case that $s_2.\text{TRY}_p \cap t_2.\text{TRY}_q = \emptyset$, we have:

$$|t_2.\text{FREE}_q \setminus t_2.\text{TRY}_q \cap \overline{s_2.\text{FREE}_p \setminus s_2.\text{TRY}_p}| \leq m(q-p) + m - 1 \tag{1}$$

Moreover, since $s_2.\text{FREE}_p \setminus s_2.\text{TRY}_p \subseteq \text{A}$ and $|s_2.\text{FREE}_p \setminus s_2.\text{TRY}_p| \geq \beta \geq 3m^2$, $|\text{A}| \geq 3m^2$. Similarly $|\text{B}| \geq 3m^2$. We have:

$$(q-1)\frac{|\text{B}|}{m} = (p-1)\frac{|\text{B}|}{m} + (q-p)\frac{|\text{B}|}{m} > (p-1)\frac{|\text{A}|}{m} + (q-p)\frac{|\text{B}|}{m} \Rightarrow$$

$$\Rightarrow (q-1)\frac{|\text{B}|}{m} > (p-1)\frac{|\text{A}|}{m} + 3m(q-p) \Rightarrow$$

$$\Rightarrow (q-1)\frac{|\text{B}|}{m} > (p-1)\frac{|\text{A}|}{m} + (3m-1)(q-p) + (q-p) \Rightarrow$$

$$\Rightarrow (q-1)\frac{|\text{B}|}{m} > (p-1)\frac{|\text{A}|}{m} + (3m-1)(q-p) + \frac{(q-p)(m-1)}{m} \Rightarrow$$

$$\left\lfloor (q-1)\frac{|\text{B}| - (m-1)}{m} \right\rfloor + 1 \geq \left\lfloor (p-1)\frac{|\text{A}| - (m-1)}{m} \right\rfloor + 1 + (3m-1)(q-p) \tag{2}$$

Since $s_2'.\text{NEXT}_p = t_2'.\text{NEXT}_q = i$, we have:

$$[i]_{s_2.\text{FREE}_p \setminus s_2.\text{TRY}_p} = \left\lfloor (p-1)\frac{|\text{A}| - (m-1)}{m} \right\rfloor + 1$$

$$[i]_{t_2.\text{FREE}_q \setminus t_2.\text{TRY}_q} = \left\lfloor (q-1)\frac{|\text{B}| - (m-1)}{m} \right\rfloor + 1$$

Equation 2 becomes:

$$[i]_{t_2.\text{FREE}_q \setminus t_2.\text{TRY}_q} \geq [i]_{s_2.\text{FREE}_p \setminus s_2.\text{TRY}_p} + (3m-1)(q-p)$$

Thus set $t_2.\text{FREE}_q \setminus t_2.\text{TRY}_q$ must have at least $(3m-1)(q-p)$ more elements with rank less that the rank of i, than set $s_2.\text{FREE}_p \setminus s_2.\text{TRY}_p$ does. This is a contradiction since from eq. 1 we have that:

$$|t_2.\text{FREE}_q \setminus t_2.\text{TRY}_q \cap \overline{s_2.\text{FREE}_p \setminus s_2.\text{TRY}_p}| \leq m(q-p) + m - 1$$

Case 2 $|\text{B}| \leq |\text{A}|$**:** We have that $\left|\overline{\text{A}} \cap \text{B}\right| \leq (q-p)\cdot m$ and $\left|\text{A} \cap \overline{\text{B}}\right| \leq (q-p)\cdot m$ from the contradiction assumption. Since $s_2.\text{FREE}_p \setminus s_2.\text{TRY}_p$ can have up to $m-1$ less elements than A – the elements of set $s_2.\text{TRY}_p$ – and it can be the case that $s_2.\text{TRY}_p \cap t_2.\text{TRY}_p = \emptyset$, we have:

$$|t_2.\text{FREE}_q \setminus t_2.\text{TRY}_q \cap \overline{s_2.\text{FREE}_p \setminus s_2.\text{TRY}_p}| \leq m(q-p) + m - 1 \tag{3}$$

From the contradiction assumption and the case 2 assumption we have that $|\text{B}| \leq |\text{A}| \leq |\text{B}| + (q-p)\cdot m$. Moreover $|\text{A}| \geq \beta \geq 3m^2$ and $|\text{B}| \geq \beta \geq 3m^2$. We have:

$$(q-1)\frac{|\text{B}| + (q-p)m}{m} = (p-1)\frac{|\text{B}| + (q-p)m}{m} + (q-p)\frac{|\text{B}| + (q-p)m}{m} \geq$$

$$\geq (p-1)\frac{|\mathrm{A}|}{m}+(q-p)\frac{|\mathrm{B}|+(q-p)m}{m} \geq (p-1)\frac{|\mathrm{A}|}{m}+3m(q-p)+(q-p)^2 \Rightarrow$$

$$\Rightarrow (q-1)\frac{|\mathrm{B}|}{m} \geq (p-1)\frac{|\mathrm{A}|}{m}+3m(q-p)+(q-p)^2-(q-1)(q-p) \Rightarrow$$

$$\Rightarrow (q-1)\frac{|\mathrm{B}|}{m} \geq (p-1)\frac{|\mathrm{A}|}{m}+(3m-p+1)(q-p) \Rightarrow$$

$$\Rightarrow (q-1)\frac{|\mathrm{B}|}{m} \geq (p-1)\frac{|\mathrm{A}|}{m}+(2m+2)(q-p) \Rightarrow$$

$$\Rightarrow (q-1)\frac{|\mathrm{B}|}{m} \geq (p-1)\frac{|\mathrm{A}|}{m}+(2m+1)(q-p)+\frac{(q-p)(m-1)}{m} \Rightarrow$$

$$\left\lfloor (q-1)\frac{|\mathrm{B}|-(m-1)}{m}\right\rfloor+1 \geq \left\lfloor (p-1)\frac{|\mathrm{A}|-(m-1)}{m}\right\rfloor+1+(2m+1)(q-p) \qquad (4)$$

Since $s_2'.\mathrm{NEXT}_p = t_2'.\mathrm{NEXT}_q = i$, we have:

$$[i]_{s_2.\mathrm{FREE}_p \setminus s_2.\mathrm{TRY}_p} = \left\lfloor (p-1)\frac{|\mathrm{A}|-(m-1)}{m}\right\rfloor+1$$

$$[i]_{t_2.\mathrm{FREE}_q \setminus t_2.\mathrm{TRY}_q} = \left\lfloor (q-1)\frac{|\mathrm{B}|-(m-1)}{m}\right\rfloor+1$$

Equation 4 becomes:

$$[i]_{t_2.\mathrm{FREE}_q \setminus t_2.\mathrm{TRY}_q} \geq [i]_{s_2.\mathrm{FREE}_p \setminus s_2.\mathrm{TRY}_p} + (2m+1)(q-p)$$

Thus set $t_2.\mathrm{FREE}_q \setminus t_2.\mathrm{TRY}_q$ must have at least $(2m+1)(q-p)$ more elements with rank less that the rank of i, than set $s_2.\mathrm{FREE}_p \setminus s_2.\mathrm{TRY}_p$. This is a contradiction since from eq. 3 we have that:

$$|t_2.\mathrm{FREE}_q \setminus t_2.\mathrm{TRY}_q \cap \overline{s_2.\mathrm{FREE}_p \setminus s_2.\mathrm{TRY}_p}| \leq m(q-p)+m-1$$

□

Next we show that if a process p detects consecutive collisions with process q, the processes p,q attempted to perform the jobs associated with the collisions in the same order and the order process p detects the collisions according to Definition 5.2 is the same as the order processes p,q attempted to perform the jobs.

In the proofs that follow, for a state s in execution α we define as s.DONE the following set: $s.\text{DONE} = \{i \in \mathcal{J} | \exists p \in \mathcal{P} \text{ and } j \in \{1,\ldots,n\} : s.done_p(j) = i\}$.

Lemma 5.6. *In an execution $\alpha \in execs(\text{KK}_\beta)$ for any $\beta \geq m$ if there exist processes p,q, jobs $i_1,i_2 \in \mathcal{J}$ and states $\tilde{s}_1 < \tilde{s}_2$ such that process p collided with process q in job i_1 at state $\tilde{s}_1$ and in job i_2 at state $\tilde{s}_2$ according to Definition 5.2, then there exist transitions $\left(s_1, \mathsf{compNext}_p, s_1'\right) \mapsto \tilde{s}_1$, $\left(s_2, \mathsf{compNext}_p, s_2'\right) \mapsto \tilde{s}_2$ and $\left(t_1, \mathsf{compNext}_q, t_1'\right)$, $\left(t_2, \mathsf{compNext}_q, t_2'\right)$ where $s_1'.\text{NEXT}_p = t_1'.\text{NEXT}_q = i_1$, $s_2'.\text{NEXT}_p = t_2'.\text{NEXT}_q = i_2$, $s_1'.\text{STATUS}_p = s_2'.\text{STATUS}_p = t_1'.\text{STATUS}_q = t_2'.\text{STATUS}_q = set\_next$ such that:*

$$s_1 < s_2 \text{ and } t_1 < t_2 \,.$$

Proof. From Definition 5.2 we have that there exist transitions $\left(s_1, \mathsf{compNext}_p, s_1'\right)$, $\left(s_2, \mathsf{compNext}_p, s_2'\right)$ with $s_1'.\text{NEXT}_p = i_1$, $s_2'.\text{NEXT}_p = i_2$, $s_1'.\text{STATUS}_p = s_2'.\text{STATUS}_p = set\_next$, and there exists no action $\pi_1 = \mathsf{compNext}_p$ for which $s_1 < \pi_1 < \tilde{s}_1$ or $s_2 < \pi_1 < \tilde{s}_2$. From the latter and the fact that $\tilde{s}_1 < \tilde{s}_2$, it must be the case that $s_1 < \tilde{s}_1 < s_2 < \tilde{s}_2$. Furthermore from Definition 5.2 we have that there exist transitions $\left(t_1, \mathsf{compNext}_q, t_1'\right)$, $\left(t_2, \mathsf{compNext}_q, t_2'\right)$ with $t_1'.\text{NEXT}_q = i_1$, $t_2'.\text{NEXT}_q = i_2$, $t_1'.\text{STATUS}_q = t_2'.\text{STATUS}_q = set\_next$, such that $t_1' < \tilde{s}_1$ and $t_2' < \tilde{s}_2$. We can pick those transitions in α in such a way that there exists no other transition between t_1' and $\tilde{s}_1$ that sets NEXT_q to i_1 and similarly there exists no other transition between t_2' and $\tilde{s}_2$ that sets NEXT_q to i_2. We need to prove now that $t_1 < t_2$. We will prove this by contradiction.

Let $t_2 < t_1$. Since $t_1^{'} < \tilde{s}_1$, we have that $t_2 < t_1 < t_1^{'} < \tilde{s}_1 < s_2 < \tilde{s}_2$. Since from Definition 5.2 either $\tilde{s}_1.next_q = i_1$ or there exists $j \in \{1,\ldots,n\}$ such that $\tilde{s}_1.done_{q,j} = i_1$, it must be the case that $\tilde{s}_2.\text{STATUS}_p = gather\_done$, $\tilde{s}_2.\text{Q}_p = q$ and there exists $j' \in \{1,\ldots,n\}$ such that $\tilde{s}_2.done_{q,j'} = i_2$. Essentially, it must be that case that process q performed job i_2 after transition $\left(t_2, \mathsf{compNext}_q, t_2^{'}\right)$. This means that there exists transition $\left(t_3, \mathsf{done}_q, t_3^{'}\right)$ and $j' \in \{1,\ldots,n\}$ such that $t_3^{'}.done_{q,j'} = i_2$ and $t_2 < t_3^{'} < t_1 < t_1^{'} < \tilde{s}_1 < s_2 < \tilde{s}_2$.

If $\tilde{s}_1.\text{STATUS}_p = gather\_try$ then from algorithm KK_β we have that $\tilde{s}_1.\text{DONE} \subseteq s_2.\text{DONE}_p$, since actions $\mathsf{gatherTry}_p$ are followed by actions $\mathsf{gatherDone}_p$ before any action $\mathsf{setNext}_p$ takes place. As a result $i_2 \in s_2.\text{DONE}_p$, which is a contradiction since $\left(s_2, \mathsf{compNext}_p, s_2^{'}\right) \notin trans(\text{KK}_\beta)$ if $i_2 \in s_2.\text{DONE}_p$ and $s_2^{'}.\text{NEXT}_p = i_2$, $s_2^{'}.\text{STATUS}_p = set\_next$.

If $\tilde{s}_1.\text{STATUS}_p = gather\_done$ then from algorithm KK_β we have that $\tilde{s}_1.\text{Q}_p = q$ and there exists $j \in \{1,\ldots,n\}$ such that $\tilde{s}_1.\text{POS}_p(q) = j$ and $\tilde{s}_1.done_{q,j} = i_1$. Since $t_2 < t_3^{'} < t_1 < t_1^{'} < \tilde{s}_1 < s_2 < \tilde{s}_2$ it must be the case that $j' < j$ and as a result $i_2 \in \tilde{s}_1.\text{DONE}_p$. Clearly $\tilde{s}_1.\text{DONE}_p \subseteq s_2.\text{DONE}_p$, which is a contradiction since $\left(s_2, \mathsf{compNext}_p, s_2^{'}\right) \notin trans(\text{KK}_\beta)$ if $i_2 \in s_2.\text{DONE}_p$ and $s_2^{'}.\text{NEXT}_p = i_2$, $s_2^{'}.\text{STATUS}_p = set\_next$. □

Next we show that if two consecutive collisions take place between processes p,q, and p detects the one collision and q the other, the processes p,q attempted to perform the jobs associated with the collisions in the same order and the order in which the processes detect the collisions according to Definition 5.2 is the same as the order the processes p,q attempted to perform the jobs.

Lemma 5.7. *In an execution $\alpha \in execs(\text{KK}_\beta)$ for any $\beta \geq m$ if there exist processes p,q, jobs $i_1,i_2 \in \mathcal{J}$ and states $\tilde{s}_1 < \tilde{s}_2$ such that process p collided with process q in job i_1 at state $\tilde{s}_1$ and process q collided with process p in job i_2 at state $\tilde{s}_2$ according to Definition 5.2, then there exist transitions $\left(s_1, \mathsf{compNext}_p, s_1^{'}\right) \mapsto \tilde{s}_1$, $\left(t_2, \mathsf{compNext}_q, t_2^{'}\right) \mapsto \tilde{s}_2$ and*

$\left(s_2, \mathsf{compNext}_p, s_2^{'}\right)$, $\left(t_1, \mathsf{compNext}_q, t_1^{'}\right)$, *where* $s_1^{'}.\text{NEXT}_p = t_1^{'}.\text{NEXT}_q = i_1$, $s_2^{'}.\text{NEXT}_p = t_2^{'}.\text{NEXT}_q = i_2$, $s_1^{'}.\text{STATUS}_p = s_2^{'}.\text{STATUS}_p = t_1^{'}.\text{STATUS}_q = t_2^{'}.\text{STATUS}_q =$ *set_next such that:*

$$s_1 < s_2 \text{ and } t_1 < t_2 \, .$$

Proof. From Definition 5.2 we have that there exist transitions $\left(s_1, \mathsf{compNext}_p, s_1^{'}\right)$, $\left(s_2, \mathsf{compNext}_p, s_2^{'}\right)$ with $s_1^{'}.\text{NEXT}_p = i_1$, $s_2^{'}.\text{NEXT}_p = i_2$, $s_1^{'}.\text{STATUS}_p = s_2^{'}.\text{STATUS}_p =$ *set_next*, and there exists no action $\pi_1 = \mathsf{compNext}_p$ for which $s_1 < \pi_1 < \tilde{s}_1$. Furthermore from Definition 5.2 we have that there exist transitions $\left(t_1, \mathsf{compNext}_q, t_1^{'}\right)$, $\left(t_2, \mathsf{compNext}_q, t_2^{'}\right)$ with $t_1^{'}.\text{NEXT}_q = i_1$, $t_2^{'}.\text{NEXT}_q = i_2$, $t_1^{'}.\text{STATUS}_q = t_2^{'}.\text{STATUS}_q =$ *set_next*, and there exists no action $\pi_2 = \mathsf{compNext}_q$ for which $t_2 < \pi_2 < \tilde{s}_2$. From the later and the fact that $\tilde{s}_1 < \tilde{s}_2$, it must be the case that $t_1 < t_2 < \tilde{s}_2$. We can pick the transitions that are enabled by states t_1 and s_2 in α in such a way that there exists no other transition between $t_1^{'}$ and $\tilde{s}_1$ that sets NEXT_q to i_1 and similarly there exists no other transition between $s_2^{'}$ and $\tilde{s}_2$ that sets NEXT_p to i_2. We need to prove now that $s_1 < s_2$. We will prove this by contradiction.

Let $s_2 < s_1$. From algorithm KK_β and Definition 5.2 there exist transitions $\left(s_3, \mathsf{setNext}_p, s_3^{'}\right)$, and $\left(t_3, \mathsf{setNext}_q, t_3^{'}\right)$, where $s_3^{'}.next_p = i_2$, $t_3^{'}.next_q = i_1$ and $s_2 < s_3^{'} < s_1$, $t_1 < t_3^{'} < t_2$. There are 2 cases, either $s_3^{'} < t_3^{'}$ or $t_3^{'} < s_3^{'}$.

Case 1 $s_3^{'} < t_3^{'}$**:** We have that $s_3^{'} < t_3^{'} < t_2$ and $\left(t_2, \mathsf{compNext}_q, t_2^{'}\right)$, where $t_2^{'}.\text{NEXT}_q = i_2$ and $t_2^{'}.\text{STATUS}_q =$ *set_next* which means that $i_2 \notin t_2.\text{TRY}_q \cup t_2.\text{DONE}_q$. This is a contradiction since the $t_2.\text{TRY}_q$ and $t_2.\text{DONE}_q$ are computed by actions $\mathsf{gatherTry}_q$ and $\mathsf{gatherDone}_q$ that are preceded by state $s_3^{'}$. Either $i_2 \in t_2.\text{TRY}_q$ or a new action $\mathsf{setNext}_p$ took place before the $\mathsf{gatherTry}_q$ actions. In the latter case, if there is a transition $\left(s_4, \mathsf{done}_p, s_4^{'}\right)$, where $s_4.next_p = i_2$, before the action $\mathsf{setNext}_p$, it must be the case that

$i_2 \in t_2.\text{DONE}_q$. If there exists no such transition we have again a contradiction since we cannot have a collision in job i_2 at state $\tilde{s}_2$ as defined in Definition 5.2.

Case 2 $t_3^{'} < s_3^{'}$**:** We have that $t_3^{'} < s_3^{'} < s_1$ and $\left(s_1, \mathsf{compNext}_p, s_1^{'}\right)$, where $s_1^{'}.\text{NEXT}_p = i_1$ and $s_1^{'}.\text{STATUS}_p = \textit{set\_next}$ which means that $i_1 \notin s_1.\text{TRY}_p \cup s_1.\text{DONE}_p$. This is a contradiction since the $s_1.\text{TRY}_p$ and $s_1.\text{DONE}_p$ sets are computed by $\mathsf{gatherTry}_p$ and $\mathsf{gatherDone}_p$ actions that are preceded by state $t_3^{'}$. Either $i_1 \in s_1.\text{TRY}_p$ or a new action $\mathsf{setNext}_q$ took place before the $\mathsf{gatherTry}_p$ actions. In the latter case, if there is a transition $\left(t_4, \mathsf{done}_q, t_4^{'}\right)$, where $t_4.next_q = i_1$, before the action $\mathsf{setNext}_q$, it must be the case that $i_1 \in s_1.\text{DONE}_p$. If there exists no such transition we have again a contradiction since we cannot have a collision in job i_1 at state $\tilde{s}_1$ as defined in Definition 5.2. □

Next we show that if 2 processes $p, q \in \mathcal{P}$ *collide* three times, their DONE sets at the third collision will contain at least $m \cdot (q - p)$ more jobs than they did at the first collision. This will allow us to find an upper bound on the collisions a process may participate in. It is possible that both processes become aware of a collision or only one of them does while the other one successfully completes the job.

Lemma 5.8. *If $\beta \geq 3m^2$ and in an execution $\alpha \in execs(\text{KK}_\beta)$ there exist processes $p \neq q$, jobs $i_1, i_2, i_3 \in \mathcal{J}$ and states $\tilde{s}_1 < \tilde{s}_2 < \tilde{s}_3$ such that process p, q collide in job i_1 at state $\tilde{s}_1$, in job i_2 at state $\tilde{s}_2$ and in job i_3 at state $\tilde{s}_3$ according to Definition 5.3, then there exist states $s_1 < s_3$ and $t_1 < t_3$ such that:*

$$s_1.\text{DONE}_p \cup t_1.\text{DONE}_q \subseteq s_3.\text{DONE}_p \cap t_3.\text{DONE}_q$$

$$\left|s_3.\text{DONE}_p \cup t_3.\text{DONE}_q\right| - \left|s_1.\text{DONE}_p \cup t_1.\text{DONE}_q\right| \geq m \cdot |q - p|$$

Proof. From Definitions 5.2, 5.3 we have that there exist transitions $\left(s_1, \mathsf{compNext}_p, s_1^{'}\right)$,

$\left(s_2, \mathsf{compNext}_p, s_2^{'}\right)$, $\left(s_3, \mathsf{compNext}_p, s_3^{'}\right)$ and $\left(t_1, \mathsf{compNext}_q, t_1^{'}\right)$, $\left(t_2, \mathsf{compNext}_q, t_2^{'}\right)$, $\left(t_3, \mathsf{compNext}_q, t_3^{'}\right)$, where $s_1^{'}.\text{NEXT}_p = t_1^{'}.\text{NEXT}_q = i_1$, $s_2^{'}.\text{NEXT}_p = t_2^{'}.\text{NEXT}_q = i_2$, $s_3^{'}.\text{NEXT}_p = t_3^{'}.\text{NEXT}_q = i_3$, $s_1^{'}.\text{STATUS}_p = s_2^{'}.\text{STATUS}_p = s_3^{'}.\text{STATUS}_p = t_1^{'}.\text{STATUS}_q = t_2^{'}.\text{STATUS}_q = t_3^{'}.\text{STATUS}_q = \mathit{set\_next}$ and $s_1 < \tilde{s}_1$, $t_1 < \tilde{s}_1$, $s_2 < \tilde{s}_2$, $t_2 < \tilde{s}_2$, and $s_3 < \tilde{s}_3$, $t_3 < \tilde{s}_3$. We pick from α the transitions $\left(s_1, \mathsf{compNext}_p, s_1^{'}\right)$, $\left(t_1, \mathsf{compNext}_q, t_1^{'}\right)$, in such a way that there exists no other $\mathsf{compNext}_p$, $\mathsf{compNext}_q$ between states s_1, $\tilde{s}_1$ respectively t_1, $\tilde{s}_1$ that sets NEXT_p respectively NEXT_q to i_1. We can pick in a similar manner the transitions for jobs i_2, i_3. From Lemmas 5.6, 5.7 and Definitions 5.2, 5.3 we have that $s_1 < s_2 < s_3$ and $t_1 < t_2 < t_3$. We will first prove that:

$$s_1.\text{DONE}_p \cup t_1.\text{DONE}_q \subseteq s_3.\text{DONE}_p \cap t_3.\text{DONE}_q$$

From algorithm KK_β we have that there exists in α transitions $\left(s_4, \mathsf{setNext}_p, s_4^{'}\right)$, $\left(t_4, \mathsf{setNext}_q, t_4^{'}\right)$ with $s_4^{'}.\mathit{next}_p = i_2$, $t_4^{'}.\mathit{next}_q = i_2$ and there exist no action $\pi_1 = \mathit{compNext}_p$, such that $s_2^{'} < \pi_1 < s_4$, and no action $\pi_2 = \mathit{compNext}_q$, such that $t_2^{'} < \pi_2 < t_4$. We need to prove that $t_1 < s_4$ and $s_1 < t_4$.

We start by proving that $t_1 < s_4$. In order to get a contradiction we assume that $s_4 < t_1$. From algorithm KK_β we have that there exists in α transition $\left(t_5, \mathsf{gatherTry}_q, t_5^{'}\right)$, with $t_5.\text{Q}_q = p$, and there exists no action $\pi_2 = \mathit{compNext}_q$, such that $t_5^{'} < \pi_2 < t_2$. We have that $s_4 < t_1 < t_5^{'} < t_2$ and $i_2 \notin t_2.\text{TRY}_q \cup t_2.\text{DONE}_q$. If $t_5.\mathit{next}_p = i_2$ we have a contradiction since $i_2 \in s_2.\text{TRY}_q$. If $t_5.\mathit{next}_q \neq i_2$ there exists an action $\pi_3 = \mathsf{setNext}_p$ in α, such that $s_4 < \pi_3 < t_5$. If this $\pi_3 = \mathsf{setNext}_p$ is preceded by transition $\left(s_5, \mathsf{done}_p, s_5^{'}\right)$ with $s_5.\text{NEXT}_p = i_2$, we have a contradiction since $i_2 \in t_5.\text{DONE}$ and $t_2.\text{DONE}_q$ is computed by $\mathsf{gatherDone}_q$ actions that are preceded by state t_5, which results in $i_2 \in t_2.\text{DONE}_q$. If there exists no such transition we have again a contradiction since we cannot have a collision in job i_2 at state $\tilde{s}_2$ as defined in Definition 5.2.

The case $s_1 < t_4$ is symmetric and can be proved with similar arguments.

From the discussion above we have that $t_1 < s_4$, thus $t_1.\text{DONE}_q \subseteq s_4.\text{DONE}$. Moreover $s_3.\text{DONE}_p$ is computed by $\mathsf{gatherDone}_p$ actions that are preceded by state s_4, from which we have that $t_1.\text{DONE}_q \subseteq s_3.\text{DONE}_p$. Since $s_1 < s_3$ it holds that $s_1.\text{DONE}_p \subseteq s_3.\text{DONE}_p$, thus we have that $s_1.\text{DONE}_p \cup t_1.\text{DONE}_q \subseteq s_3.\text{DONE}_p$. From $s_1 < t_4$, with similar arguments as before, we can prove that $s_1.\text{DONE}_p \cup t_1.\text{DONE}_q \subseteq t_3.\text{DONE}_q$, which gives us that:

$$s_1.\text{DONE}_p \cup t_1.\text{DONE}_q \subseteq s_3.\text{DONE}_p \cap t_3.\text{DONE}_q$$

Now it only remains to prove that:

$$\left|s_3.\text{DONE}_p \cup t_3.\text{DONE}_q\right| - \left|s_1.\text{DONE}_p \cup t_1.\text{DONE}_q\right| > m \cdot |q-p|$$

If $p < q$ from Lemma 5.5 we have that $\left|s_3.\text{DONE}_p \cap \overline{t_3.\text{DONE}_q}\right| > (q-p)m$ or $\left|\overline{s_3.\text{DONE}_p} \cap t_3.\text{DONE}_q\right| > (q-p)m$. Since $s_1.\text{DONE}_p \cup t_1.\text{DONE}_q \subseteq s_3.\text{DONE}_p \cap t_3.\text{DONE}_q$, we have that:

$$\left|s_3.\text{DONE}_p \cup t_3.\text{DONE}_q\right| - \left|s_1.\text{DONE}_p \cup t_1.\text{DONE}_q\right| > (q-p) \cdot m$$

If $q < p$ with similar arguments we have that:

$$\left|s_3.\text{DONE}_p \cup t_3.\text{DONE}_q\right| - \left|s_1.\text{DONE}_p \cup t_1.\text{DONE}_q\right| > (p-q) \cdot m$$

Combining the above we have:

$$\left|s_3.\text{DONE}_p \cup t_3.\text{DONE}_q\right| - \left|s_1.\text{DONE}_p \cup t_1.\text{DONE}_q\right| > m \cdot |q-p|$$

□

Next we prove that a process p cannot collide with a process q more than $2\left\lceil \frac{n}{m \cdot |q-p|} \right\rceil$ times in any execution.

Lemma 5.9. *If $\beta \geq 3m^2$ there exists no execution $\alpha \in execs(\mathrm{KK}_\beta)$ at which process p collided with process q in more than $2\left\lceil \frac{n}{m|q-p|} \right\rceil$ states according to Definition 5.2.*

Proof. Let execution $\alpha \in execs(\mathrm{KK}_\beta)$ be an execution at which process p collided with process q in at least $2\left\lceil \frac{n}{m|q-p|} \right\rceil + 1$ states. Let us examine the first $2\left\lceil \frac{n}{m|q-p|} \right\rceil + 1$ such states. Let those states be $\tilde{s}_1 < \tilde{s}_2 < \ldots < \tilde{s}_{2\left\lceil \frac{n}{m|q-p|} \right\rceil} < \tilde{s}_{2\left\lceil \frac{n}{m|q-p|} \right\rceil + 1}$. From Lemma 5.6 we have that there exists states $s_1 < s_2 < \ldots < s_{2\left\lceil \frac{n}{m|q-p|} \right\rceil} < s_{2\left\lceil \frac{n}{m|q-p|} \right\rceil + 1}$ that enable the $compNext_p$ actions and states $t_1 < t_2 < \ldots < t_{2\left\lceil \frac{n}{m|q-p|} \right\rceil} < t_{2\left\lceil \frac{n}{m|q-p|} \right\rceil + 1}$ that enable the $compNext_q$ actions that lead to the collisions in states $\tilde{s}_1 < \tilde{s}_2 < \ldots < \tilde{s}_{2\left\lceil \frac{n}{m|q-p|} \right\rceil} < \tilde{s}_{2\left\lceil \frac{n}{m|q-p|} \right\rceil + 1}$. Then from Lemma 5.8 we have that $\forall i \in \left\{1, \ldots, \left\lceil \frac{n}{m|q-p|} \right\rceil \right\}$:

$$\left|s_{2i+1}.\mathrm{DONE}_p \cup t_{2i+1}.\mathrm{DONE}_q\right| - \left|s_{2i-1}.\mathrm{DONE}_p \cup t_{2i-1}.\mathrm{DONE}_q\right| > m|q-p|$$

$$\left|s_{2i+1}.\mathrm{DONE}_p \cup t_{2i+1}.\mathrm{DONE}_q\right| - \left|s_1.\mathrm{DONE}_p \cup t_1.\mathrm{DONE}_q\right| > im|q-p|$$

$$\left|s_{2i+1}.\mathrm{DONE}_p \cup t_{2i+1}.\mathrm{DONE}_q\right| > im|q-p| \tag{5}$$

From eq. 5 we have that:

$$\left|s_{2\left\lceil \frac{n}{m|q-p|} \right\rceil + 1}.\mathrm{DONE}_p \cup t_{2\left\lceil \frac{n}{m|q-p|} \right\rceil + 1}.\mathrm{DONE}_q\right| > m|q-p| \left\lceil \frac{n}{m|q-p|} \right\rceil \geq n \tag{6}$$

Equation 6 leads to a contradiction since $s_{2\left\lceil \frac{n}{m|q-p|} \right\rceil + 1}.\mathrm{DONE}_p \cup t_{2\left\lceil \frac{n}{m|q-p|} \right\rceil + 1}.\mathrm{DONE}_q \subseteq \mathcal{J}$ and $|\mathcal{J}| = n$.

□

Finally we are ready to prove the main theorem on the work complexity of algorithm KK_β for $\beta \geq 3m^2$.

Theorem 5.10. *If* $\beta \geq 3m^2$ *algorithm* KK_β *has work complexity* $W_{\mathrm{KK}_\beta} = \mathrm{O}(nm \log n \log m)$.

Proof. We start with the observation that in any execution α of algorithm KK_β, if there exists process p, job i, transition $\left(s_1, \mathsf{done}_p, s_1^{'}\right)$ and $j \in \{1, \ldots, n\}$ such that $s_1.\mathrm{POS}_p(p) = j$, $s_1.\mathrm{NEXT}_p = i$, for any process $q \neq p$ there exists at most one transition $\left(t_1, \mathsf{gatherDone}_q, t_1^{'}\right)$ in α, with $t_1.\mathrm{Q}_q = p$, $t_1.\mathrm{POS}_q(p) = j$ and $t_1 \geq s_1$. Such transition performs exactly one read operation from the shared memory, one insertion at the set DONE_q and one removal from the set FREE_q, thus such a transition costs $\mathrm{O}(\log n)$ work. Clearly there exist at most $m-1$ such transitions for each done_p. From Lemma 5.1 for all processes there can be at most n actions done_p in any execution α of algorithm KK_β. Each done_p action performs one write operation in shared memory, one insertion at the set DONE_p and one removal from the set FREE_p, thus such an action has cost $\mathrm{O}(\log n)$ work. Furthermore any done_p is preceded by $m-1$ $\mathsf{gatherTry}_p$ read actions that read the *next* array and each add at most one element to the set TRY_p with cost $\mathrm{O}(\log n)$ and $m-1$ $\mathsf{gatherDone}_p$ read actions that do not add elements in the DONE_p set. Note that we have already counted the $\mathsf{gatherDone}_p$ read actions that result in adding jobs at the DONE_p set. Finally any done_p action is preceded by one $\mathsf{compNext}_p$ action. This action is dominated by the cost of the $rank(\mathrm{FREE}_p, \mathrm{TRY}_p, i)$ function. If the sets FREE_p, TRY_p are represented with some efficient tree structure like *red-black tree* or some variant of *B-tree* [7, 22] that allows insertion, deletion and search of an element in $\mathrm{O}(\log n)$, an invocation of function $rank(\mathrm{FREE}_p, \mathrm{TRY}_p, i)$ costs $\mathrm{O}(m \log n)$ work. That gives us a total of $\mathrm{O}(nm \log n)$ work associated with the done_p actions.

If a process p collided with a process q in job i at state s, we have an extra $\mathsf{compNext}_p$

action, $m-1$ extra $\mathsf{gatherTry}_p$ read actions and insertions in the TRY_p set and $m-1$ $\mathsf{gatherDone}_p$ read actions that do not add elements in the DONE_p set. Thus each collision costs $\text{O}(m \log n)$ work. Since $\beta \geq 3m^2$ from Lemma 5.9 for two distinct processes p, q we have that in any execution α of algorithm KK_β there exist less than $2\left\lceil \frac{n}{m|q-p|} \right\rceil$ collisions. For process p if we count all such collisions with any other process q we get:

$$\sum_{q \in \mathcal{P}-\{p\}} 2\left\lceil \frac{n}{m|q-p|} \right\rceil \leq 2(m-1) + \frac{2n}{m} \sum_{q \in \mathcal{P}-\{p\}} \frac{1}{|q-p|} \leq$$

$$\leq 2(m-1) + \frac{4n}{m} \sum_{i=1}^{\lceil \frac{m}{2} \rceil} \frac{1}{i} \leq 2(m-1) + \frac{4n}{m} \log m \tag{7}$$

If we count the total number of collisions for all the m processes we get that if $\beta \geq 3m^2$ in any execution of algorithm KK_β there can be at most $2m^2 + 4n \log m < 4(n+1) \log m$ collisions (since $n > \beta$). Thus collisions cost $\text{O}(nm \log n \log m)$ work. Finally any process p that fails may add in the work complexity less than $\text{O}(m \log n)$ work from its $\mathsf{compNext}_p$ action and from reads (if the process fails without performing a done_p action after its latest $\mathsf{compNext}_p$ action). So for the work complexity of algorithm KK_β if $\beta \geq 3m^2$ we have that $W_{\text{KK}_\beta} = \text{O}(nm \log n \log m)$. □

5.4 Open Problems

There is an open question concerning the work complexity of algorithm KK_β for $\beta < 3m^2$. We expect that such an analysis will be quite involved. A good result from the work complexity of KK_β for $\beta < 3m^2$, could lead in interesting new iterative solutions for both the at-most-once problem and the write-all problem. A different direction would be to perform a probabilistic work complexity analysis for $\beta < 3m^2$ using similar techniques as the ones applied in [15].

There still exists an effectiveness gap between the shown effectiveness of $n-2m+2$ of algorithm KK_β and the known effectiveness bound of $n-m+1$. It would be interesting to see if this can be bridged for wait-free deterministic algorithms. If not, it would be interesting to see deterministic algorithms that solve the problem when some progress requirements are met.

Algorithm KK_β has a $\mathrm{O}(nm\log n)$ space complexity for shared memory. By using a linked list instead of a matrix in which processes announce the jobs they complete, algorithm KK_β can be modified to have $\mathrm{O}(n\log n)$ shared memory space complexity. The correctness analysis of the modified algorithm will be quite involved, since now the cells of the linked list can be potentially written by any process, and the correctness of the implementation relies in the at-most-once property of job execution.

Algorithm KK_β can be readily modified, in order to provide a solution for a dynamic setting or streaming setting, where jobs are not known a priory and may arrive indefinitely. We conjecture that if infinite jobs arrive, then infinite jobs will be performed, and that a process failure may block exactly 1 job for infinite steps, while all other jobs will be eventually executed provided that more jobs arrive. It would be interesting to reduce the space requirements in such a setting, in order to make the solution practical.

Finally, a different direction would be to examine whether algorithm KK_β can be modified in order to provide a k-adaptive deterministic solution for the at-most-once problem.

6 Iterative Algorithms Based on KK_β

In this section we present, analyze and prove correct two asynchronous deterministic algorithms called IterativeKK(ε) and WA_IterativeKK(ε). The first algorithm solves the At-Most-Once problem, while the second solves the Write-All problem. The algorithms are presented and analyzed using pseudocode. Both algorithms are iterative and use algo-

rithm KK_β as a building block.

6.1 An Asymptotically Work Optimal Iterative Algorithm for the At-Most-Once

We demonstrate how to solve the at-most-once problem with effectiveness $n - \text{O}(m^2 \log n \log m)$ and work complexity $\text{O}(n + m^{(3+\varepsilon)} \log n)$, for any constant $\varepsilon > 0$, such that $1/\varepsilon$ is a positive integer, when $m = \text{O}(\sqrt[3]{n})$, using algorithm KK_β with $\beta = 3m^2$. Algorithm IterativeKK(ε), presented in Fig. 5, performs iterative calls to a variation of algorithm KK_β, called IterStepKK. IterativeKK(ε) has $3 + 1/\varepsilon$ distinct matrices *done* and vectors *next* in shared memory, with different granularities. One *done* matrix, stores the regular jobs performed, while the remaining $2 + 1/\varepsilon$ matrices store *super-jobs*. Super-jobs are groups of consecutive jobs. From them, one stores super-jobs of size $m \log n \log m$, while the remaining $1 + 1/\varepsilon$ matrices, store super-jobs of size $m^{1-i\varepsilon} \log n \log^{1+i} m$ for $i \in \{1, \ldots, 1/\varepsilon\}$. The $3 + 1/\varepsilon$ distinct vectors *next* are used in a similar way as the matrices *done*.

The algorithm IterStepKK is different from KK_β in the following ways. First, all instances of IterStepKK work for $\beta = 3m^2$. Moreover, IterStepKK has a termination flag in shared memory. This termination flag is initially 0 and is set to 1 by any process that decides to terminate. In the execution of algorithm IterStepKK, a process p, that in an action $\mathsf{compNext}_p$ has $|\text{FREE}_p \setminus \text{TRY}_p| < 3m^2$, sets the termination flag to 1, computes new sets FREE_p and TRY_p, returns the set $\text{FREE}_p \setminus \text{TRY}_p$ and terminates. After a process p checks if it is safe to perform a job, the process also checks the termination flag and if the flag is 1, the process instead of performing the job, computes new sets FREE_p and TRY_p, returns the set $\text{FREE}_p \setminus \text{TRY}_p$ and terminates. Finally, algorithm IterStepKK takes

IterativeKK(ε) **for process** p**:**

00 $\text{size}_{p,1} \leftarrow 1$

01 $\text{size}_{p,2} \leftarrow m \log n \log m$

02 $\text{FREE}_p \leftarrow \text{map}\left(\mathcal{J}, \text{size}_{p,1}, \text{size}_{p,2}\right)$

03 $\text{FREE}_p \leftarrow \text{IterStepKK}\left(\text{FREE}_\text{p}, \text{size}_{\text{p},2}\right)$

04 **for**$(i \leftarrow 1, i \leq 1/\varepsilon, i++)$

05 $\quad \text{size}_{p,1} \leftarrow \text{size}_{p,2}$

06 $\quad \text{size}_{p,2} \leftarrow m^{1-i\varepsilon} \log n \log^{1+i} m$

07 $\quad \text{FREE}_p \leftarrow \text{map}\left(\text{FREE}_p, \text{size}_{p,1}, \text{size}_{p,2}\right)$

08 $\quad \text{FREE}_p \leftarrow \text{IterStepKK}\left(\text{FREE}_\text{p}, \text{size}_{\text{p},2}\right)$

09 **endfor**

10 $\text{size}_{p,1} \leftarrow \text{size}_{p,2}$

11 $\text{size}_{p,2} \leftarrow 1$

12 $\text{FREE}_p \leftarrow \text{map}\left(\text{FREE}_p, \text{size}_{p,1}, \text{size}_{p,2}\right)$

13 $\text{FREE}_p \leftarrow \text{IterStepKK}\left(\text{FREE}_\text{p}, \text{size}_{\text{p},2}\right)$

Figure 5: Algorithm IterativeKK(ε): pseudocode

as inputs the variable *size* and a set SET_1, such that $|\text{SET}_1| > 3m^2$, and returns the set SET_2 as output. SET_1 contains super-jobs of size *size*. In IterStepKK, with an action $\text{do}_{p,j}$ process p performs all the jobs of super-job j. A process p performs as many super-jobs as it can and returns in SET_2 the super-jobs it can verify that no process will perform.

In algorithm IterativeKK(ε) we use also the function $\text{SET}_2 = \text{map}\left(\text{SET}_1, \text{size}_1, \text{size}_2\right)$, that takes the set of super-jobs SET_1, with super-jobs of size $size_1$ and maps it to a set of super-jobs SET_2 with size $size_2$. A job i is always mapped to the same super-job of a specific size and there is no intersection between the jobs in super-jobs of the same size.

6.1.1 Analysis of algorithm IterativeKK(ε)

We begin the analysis of algorithm IterativeKK(ε) by showing in Theorem 6.3 that IterativeKK(ε) solves the at-most-once problem. This is done by first showing in Lemma 6.1 that algorithm IterStepKK solves the at-most-once problem for the set of all super-jobs of a specific size, and then by showing in Lemma 6.2 that there exist no per-

formed super-jobs in any output set SET_2. We complete the analysis with Theorem 6.4, where we show that algorithm IterativeKK(ε) has effectiveness $E_{\text{IterativeKK}(\varepsilon)}(n,m,f) = n - \text{O}(m^2 \log n \log m)$ and work complexity $W_{\text{IterativeKK}(\varepsilon)} = \text{O}(n + m^{3+\varepsilon} \log n)$.

Let the set of all super-jobs of a specific size d be SuperSet_d. All invocations of algorithm IterStepKK on sets $\text{SET}_1 \subseteq \text{SuperSet}_d$, use the matrix *done* and vector *next* that correspond to the super-jobs of size d. Moreover each process p invokes algorithm IterStepKK for a set $\text{SET}_1 \subseteq \text{SuperSet}_d$ only once. We have the following lemma.

Lemma 6.1. *Algorithm* IterStepKK *solves the at-most-once problem for the set* SuperSet_d.

Proof. As described above, algorithm IterStepKK is different from KK_β in the following ways:

- Process p, on algorithm IterStepKK, has an input set $\text{SET}_1 \subseteq \text{SuperSet}_d$ of super-jobs of size d to be performed and outputs a set $\text{SET}_2 \subset \text{SuperSet}_d$ of super-jobs, that have not been performed. Process p initially sets its set FREE_p, equal to SET_1 and proceeds as it would do when executing KK_β, with the difference that an action $\text{do}_{p,i}$ results in performing all the jobs under super-job i. Entries in the matrix *done* and vector *next* in shared memory correspond to the identifiers of super-jobs of set SuperSet_d. Again after its initialization, entries are only removed from set FREE_p.

  Note that the main difference caused by this modification, between algorithm IterStepKK and algorithm KK_β, is that jobs are replaced by super-jobs, and that the initial sets FREE_p and FREE_q of processes p, q could be set to different subsets of set SuperSet_d. This does not affect the correctness of the algorithm, since in any state s of an execution α of algorithm KK_β, the sets FREE_p and FREE_q could be different subsets of the set of all jobs $\mathcal{J}$.

- Algorithm IterStepKK has a termination flag in shared memory. The termination flag is initially 0 and is set to 1 by any process that decides to terminate. As mentioned above, any process that discovers that $|\text{FREE}_p \setminus \text{TRY}_p| < 3m^2$ in an action $\mathsf{compNext}_p$, sets the termination flag to 1, computes new sets FREE_p and TRY_p, returns the set $\text{FREE}_p \setminus \text{TRY}_p$ and terminates. This modification only affects the sequence of actions during the termination of a process p. Observe process p does not perform any super-jobs in that termination sequence.

  Additionally, after a process p checks if it is safe to perform a super-job, it also checks the termination flag and if the flag is 1, the process instead of performing the super-job, enters the termination sequence, computing new sets FREE_p and TRY_p, returning the set $\text{FREE}_p \setminus \text{TRY}_p$ and terminating. A process p first checks if it is safe to perform a super-job according to algorithm KK_β and then checks the flag. Thus this modification only affects the effectiveness, but not the correctness of the algorithm, since it could only result in a super-job that was safe to perform not being performed.

- Finally all instances of IterStepKK work for $\beta = 3m^2$. This does not affect correctness, since Lemma 5.1 holds for any β.

It is easy to see that none of the modifications described above affect the key arguments in the proof of Lemma 5.1. Thus with similar arguments as in the proof of Lemma 5.1, we can show that there exists no execution of algorithm IterStepKK, where two distinct actions $\pi = \mathsf{do}_{p,i}$ and $\pi' = \mathsf{do}_{q,i}$ take place for a super-job $i \in \text{SuperSet}_d$ and processes $p, q \in \mathcal{P}$ (p could be equal to q). □

Next we show that in the output sets of algorithm IterStepKK at a specific iteration (calls for super-jobs of size d), no completed super-jobs are included. Combined with

the previous lemma, this argument will help us establish that algorithm IterativeKK(ε) solves that at-most-once problem.

Lemma 6.2. *There exists no execution α of algorithm* IterStepKK*, such that there exists action $\mathsf{do}_{q,i} \in \alpha$ for some process q and super-job i in the output set $\text{SET}_2 \subset \text{SuperSet}_d$ of some process p (p could be equal to process q).*

Proof. As described above, a process p before terminating algorithm IterStepKK, either sets the flag to 1 or observes that the flag is set to 1. The process p then computes new sets FREE_p and TRY_p, returns the set $\text{FREE}_p \setminus \text{TRY}_p$ and terminates its execution of algorithm IterStepKK for input set $\text{SET}_1 \subseteq \text{SuperSet}_d$ and super-jobs of size d. Let state s be the state at which process p terminates, we have that $\text{SET}_2 = s.\text{FREE}_p \setminus s.\text{TRY}_p$. If $p = q$ and there exists action $\pi = \mathsf{do}_{p,i}$ in execution α of algorithm IterStepKK, for super-jobs $i \in \text{SuperSet}_d$, clearly $\pi < s$, from which we have that $i \notin s.\text{FREE}_p$ and thus $i \notin \text{SET}_2$.

It is easy to see that if $p \neq q$ and $i \in \text{SET}_2$ of process p, there exists no action $\pi = \mathsf{do}_{q,i}$ in execution α. If $i \in \text{SET}_2$ then $i \in s.\text{FREE}_p$ and $i \notin s.\text{TRY}_p$. Moreover process p either set flag to 1 or observed that the flag was set, before computing sets $s.\text{FREE}_p$ and $s.\text{TRY}_p$. If there exists $\pi = \mathsf{do}_{q,i} \in \alpha$, for process q, it must be the case that after process q performed the transition $\left(t, \mathsf{compNext}_q, t'\right) \mapsto \pi$ (see Definition 5.1 of immediate predecessor), it read the flag and found it was equal to 0. This leads to a contradiction, since it must be the case that either $i \in s.\text{TRY}_p$ or $i \notin s.\text{FREE}_p$. □

We are ready now to show the correctness of algorithm IterativeKK(ε).

Theorem 6.3. *Algorithm* IterativeKK(ε) *solves the at-most-once problem.*

Proof. From Lemma 6.1 we have that any super-job of a specific size d is performed at-most-once (if performed at all) in the execution of algorithm IterStepKK for the super-

jobs in the set SuperSet_d. Moreover, from Lemma 6.2 we have that super-jobs in the output sets of an execution of algorithm IterStepKK for super-jobs of size d, have not been performed. Function $\text{SET}_2 = \text{map}(\text{SET}_1, \text{size}_1, \text{size}_2)$ maps the jobs in the super-jobs of set SET_1, to super-jobs in SET_2. A job i is always mapped to the same super-job of a specific size d and there is no intersection between the jobs of the super-jobs in set SuperSet_d. It is easy to see that there exists no execution of algorithm IterativeKK(ε), where a job i is performed more than once. □

We complete the analysis of algorithm IterativeKK(ε) with Theorem 6.4, which gives upper bounds for the effectiveness and work complexity of the algorithm.

Theorem 6.4. *Algorithm* IterativeKK(ε) *has* $W_{\text{IterativeKK}(\varepsilon)} = \text{O}(n + m^{3+\varepsilon} \log n)$ *work complexity and effectiveness* $E_{\text{IterativeKK}(\varepsilon)}(n,m,f) = n - \text{O}(m^2 \log n \log m)$.

Proof. In order to determine the effectiveness and work complexity of algorithm IterativeKK(ε), we compute the jobs performed by and the work spent in each invocation of IterStepKK. Moreover we compute the work that the invocations to the function map() add.

The first invocation to function map() in line 02 can be completed by process p with work $\text{O}(\frac{n}{m \log n \log m} \log n)$, since process p needs to construct a tree with $\frac{n}{m \log n \log m}$ elements. This contributes for all processes $\text{O}(\frac{n}{\log m})$ work. From Theorem 5.10 we have that IterStepKK in 03 has total work $\text{O}(n + \frac{n}{m \log n \log m} m \log n \log m) = \text{O}(n)$, where the first n comes from do actions and the second term from the work complexity of Theorem 5.10. Note that we count $\text{O}(1)$ work for each normal job executed by a do action on a super-job. That means that in the invocation of IterStepKK in line 03, do actions cost $m \log n \log m$ work. Moreover from Theorem 5.4 we have effectiveness $\frac{n}{m \log n \log m} - (3m^2 + m - 2)$ on the super-jobs of size $m \log n \log m$. From the super-jobs not completed, up to $m-1$ may be contained in the TRY_p sets upon termination in line 03. Since those super-jobs are not

added (and thus are ignored) in the output FREE_p set in line 03, up to $(m-1)m\log n\log m$ jobs may not be performed by IterativeKK(ε). The set FREE_p returned by algorithm IterStepKK in line 03 has no more than $3m^2+m-2$ super-jobs of size $m\log n\log m$.

In each repetition of the loop in lines $04-09$, the map() function in line 07 constructs a FREE_p set with at most $O(m^{2+\varepsilon}/\log m)$ elements, which costs $O(m^{2+\varepsilon})$ per process p for a total of $O(m^{3+\varepsilon})$ work for all processes. Moreover each invocation of IterStepKK in line 08 costs $O(3m^3\log n\log m+m^{3+\varepsilon}\log m) < O(m^{3+\varepsilon}\log n)$ work from Theorem 5.10, where the term $3m^3\log n\log m$ is an upper bound on the work needed for the do actions on the super-jobs. From Theorem 5.4 we have that each output FREE_p set in line 08 has at most $3m^2+m-2$ super-jobs. Moreover from each invocation of IterStepKK in line 08 at most $m-1$ super-jobs are lost in TRY sets. Those account for less than $(m-1)m\log n\log m$ jobs in each iteration, since the size of the super-jobs in the iterations of the loop in lines $04-09$ is strictly less than $m\log n\log m$.

When we leave the loop in lines $04-09$, we have a FREE_p set with at most $3m^2+m-2$ super-jobs of size $\log n\log^{1+1/\varepsilon} m$, which means that in line 12 function map() will return a set FREE_p with less than $(3m^2+m-2)(\log n\log^{1+1/\varepsilon} m)$ elements that correspond to jobs and not super-jobs. This costs for all processes a total of $O(m^3\log m\log\log n\log\log m) < O(m^{3+\varepsilon}\log n)$ work, since ε is a constant. Finally we have that IterStepKK in line 13 has from Theorem 5.10 work $O(m^3\log^2 m\log\log n\log\log m) < O(m^{3+\varepsilon}\log n)$ and from Theorem 5.4 effectiveness $(3m^2+m-2)(\log n\log^{1+1/\varepsilon} m)-(3m^2+m-2)$.

If we add up all the work, we have that $W_{\text{IterativeKK}(\varepsilon)} = O(n+m^{3+\varepsilon}\log n)$ since the loop in lines $04-09$ repeats $1+1/\varepsilon$ times and ε is a constant. Moreover for the effectiveness, we have that less than or equal to $(m-1)m\log n\log m$ jobs will be lost in the TRY set at line 03. After that strictly less than $(m-1)m\log n\log m$ jobs will be lost in the TRY

sets of the iterations of the loop in lines $04-09$ and fewer than $3m^2+m-2$ jobs will be lost from the effectiveness of the last invocation of IterStepKK in line 13. Thus we have that $E_{\text{IterativeKK}(\varepsilon)}(n,m,f) = n - \text{O}(m^2 \log n \log m)$. □

For any $m = \text{O}(\sqrt[3+\varepsilon]{n/\log n})$, algorithm IterativeKK$(\varepsilon)$ is work optimal and asymptotically effectiveness optimal.

6.2 An Asymptotically Optimal Algorithm for the Write-All Problem

---

WA_IterativeKK(ε) **for process** p**:**

00 $\text{size}_{p,1} \leftarrow 1$
01 $\text{size}_{p,2} \leftarrow m \log n \log m$
02 $\text{FREE}_p \leftarrow \text{map}\left(\beta, \text{size}_{p,1}, \text{size}_{p,2}\right)$
03 $\text{FREE}_p \leftarrow \text{WA\_IterStepKK}\left(\text{FREE}_\text{p}, \text{size}_\text{p,2}\right)$
04 **for**$(i \leftarrow 1, i \leq 1/\varepsilon, i++)$
05 $\quad \text{size}_{p,1} \leftarrow \text{size}_{p,2}$
06 $\quad \text{size}_{p,2} \leftarrow m^{1-i\varepsilon} \log n \log^{1+i} m$
07 $\quad \text{FREE}_p \leftarrow \text{map}\left(\text{FREE}_p, \text{size}_{p,1}, \text{size}_{p,2}\right)$
08 $\quad \text{FREE}_p \leftarrow \text{WA\_IterStepKK}\left(\text{FREE}_\text{p}, \text{size}_\text{p,2}\right)$
09 **endfor**
10 $\text{size}_{p,1} \leftarrow \text{size}_{p,2}$
11 $\text{size}_{p,2} \leftarrow 1$
12 $\text{FREE}_p \leftarrow \text{map}\left(\text{FREE}_p, \text{size}_{p,1}, \text{size}_{p,2}\right)$
13 $\text{FREE}_p \leftarrow \text{WA\_IterStepKK}\left(\text{FREE}_\text{p}, \text{size}_\text{p,2}\right)$
14 **for**$(i \in \text{FREE}_p)$
15 $\quad \text{do}_{p,i}$
16 **endfor**

Figure 6: Algorithm WA_IterativeKK(ε): pseudocode

Based on IterativeKK(ε) we construct algorithm WA_IterativeKK(ε) Fig. 6, that solves the *Write-All* problem [25] with work complexity $\text{O}(n+m^{(3+\varepsilon)} \log n)$, for any constant $\varepsilon > 0$, such that $1/\varepsilon$ is a positive integer. From Kanellakis and Shvartsman [25]

the Write-All problem for the shared memory model, consists of: "Using m processors write 1's to all locations of an array of size n." The problem assumes that all cells of the array are initialized to 0. Algorithm WA_IterativeKK(ε) is different from IterativeKK(ε) in two ways. It uses a modified version of IterStepKK, that instead of returning the FREE$_p \setminus$ TRY$_p$ set upon termination returns the set FREE$_p$ instead. Let us name this modified version WA_IterStepKK. Moreover in WA_IterativeKK(ε) after line 13, process p, instead of terminating, executes all jobs in the set FREE$_p$. Note that since we are interested in the Write-All problem, when process p performs a job i with action do$_{p,i}$, process p just writes 1, in the $i-$th position of the Write All array $wa[1,\ldots,n]$ in shared memory.

Theorem 6.5. *Algorithm* WA_IterativeKK(ε) *solves the Write-All problem with work complexity* $W_{\text{WA\_IterativeKK}(\varepsilon)} = \mathrm{O}(n + m^{3+\varepsilon}\log n)$.

Proof. We prove this with similar arguments as in the proof of Theorem 6.4. From Theorem 5.4 after each invocation of WA_IterStepKK the output set FREE$_p$ has less than $3m^2 + m - 1$ super-jobs. The difference is that now we do not leave jobs in the TRY$_p$ sets, since we are not interested in maintaining the at-most-once property between successive invocations of algorithm WA_IterStepKK. Since after each invocation of WA_IterStepKK the output set FREE$_p$ has the same upper bound on super-jobs as in IterativeKK(ε), with similar arguments as in the proof of Theorem 6.4, we have that at line 13 the total work performed by all processes is $\mathrm{O}(n + m^{3+\varepsilon}\log n)$. Moreover from Theorem 5.4 the output FREE$_p$ set in line p has less than $3m^2 + m - 2$ jobs. This gives us for all processes a total work of $\mathrm{O}(m^3)$ for the loop in lines $14-16$. After the loop in lines $14-16$ all jobs have been performed, since we left no TRY sets behind, thus algorithm WA_IterativeKK(ε) solves the Write-All problem with work complexity $W_{\text{WA\_IterativeKK}(\varepsilon)} = \mathrm{O}(n + m^{3+\varepsilon}\log n)$. □

For any $m = O(\sqrt[3+\varepsilon]{n/\log n})$, algorithm WA_IterativeKK(ε) is work optimal.

6.3 Open Problems

It is possible to fine grain algorithm IterativeKK(ε) in order to get a tighter effectiveness of $n-(3m^2+m-2)$ without impacting work complexity considerably. Proving correctness for such a solution will be quite involved.

7 Randomized Strong At-Most-Once Algorithm RA

In this section we present and analyze an asynchronous randomized algorithm called RA that solves the Strong At-Most-Once problem. The algorithm is presented using pseudocode. We use the adaptive adversary presented in Section 2. Moreover our memory supports only atomic read/write registers. Since the Strong At-Most-Once problem has consensus number 2, we use a randomized test-and-set object as a building block. In particular we use the *RatRace* algorithm from Alistarh *et. al.* [3]. For Strong At-Most-Once solutions we are interested in the number of participating processes k and the effectiveness of the algorithm is expressed with respect to the number of tasks n and the participating processes that failed during the execution f_k (clearly $k > f_k$). We show that RA has expected work complexity $O(n+k^{2+\varepsilon}\log n)$ and effectiveness $n-f_k$.

7.1 Algorithm RA

In Algorithm RA jobs are grouped in super-jobs. Each super-job contains $\log n$ at-most-once jobs. Every job is associated with a shared memory element of matrix W. The matrix W has size $\frac{n}{\log n}\cdot(\log n+1)$. The row 0 of matrix W is associated with the $\frac{n}{\log n}$ super-jobs, while the rows $\{1,\ldots,\log n\}$ are associated with the n jobs. Super-job i consists of the

$\log n$ jobs that are are associated with elements $W[i][j]$ for all $j \in \{1, \ldots, \log n\}$. Each element of the matrix W supports a randomized wait-free atomic test-and-set operation. The jobs test-and-set operations are used in order to guarantee at-most-once semantics, while the super-jobs test-and-set operations are used to detect and resolve collisions between processes.

Processes in Algorithm RA (see Fig. 7) create intervals of super-jobs. The main idea is that every process p picks a random super-job i as a candidate starting point for its interval. Process p calls all the test-and-set operations related with the jobs grouped under i. For each test-and-set operation p wins, it performs the corresponding job. When done, it calls the test-and-set operation of the super-job i. If the procedure described above takes place in an execution α of algorithm RA, we say that process p has *performed* super-job i, or that super-job i has been *performed*. This is independent of whether process p won the test-and-set operation for super-job i. If there exists no process p that has performed super-job i in execution α, we say that the super-job i is still *available* in execution α.

If process p wins the test-and-set operation for super-job i, it marks i as the starting point of its working interval, then performs super-job $i+1$ and keeps moving, one super-job at a time in a rightward direction, until it looses a super-job test-and-set operation. As long as p wins test-and-set operation on the super-jobs it performs, it adds the super-jobs to its current interval. If it fails this means that some other process has marked the specific super-job as the beginning of its working interval. In this case, p stops working on the interval and picks randomly a new super-job in order to start a new working interval.

The key idea behind this approach is that a process p keeps expanding a working interval that started at some super-job i until it looses a test-and-set operation at some super-job j ($j > i$). This means that some other process q won that test-and-set for j and thus q continues expanding the interval that started at i. This leads to the observation that

if k processes are participating in an execution of the algorithm RA, there exist at most $k+1$ intervals of performed super-jobs, from which, at any given point of the execution, at most k are being expanded from the right end. As long as the available super-jobs are significantly more than the k processes participating in an execution of the algorithm, from the above discussion we can show (see Section 7.2) that processes that need a new random starting point, will likely be positioned far enough of the beginning or the end of the existing intervals of performed super-jobs. This results in substantial progress being done before a process working on an interval loses a test-and-set operation and thus has to start a new interval. The latter will allow us to show that if a process has an outdated estimation of the available super-jobs, by colliding with large intervals of performed super-jobs, it will be able to learn fast about super-jobs that have been completed.

Next we present the shared memory variables, internal variables and the steps a process p has to take in algorithm RA in more detail:

Shared Variables. Algorithm RA uses matrix W of size $\frac{n}{\log n}$ x $\log n + 1$. The matrix is stored in shared memory. Each element of the matrix is initially set to 0 and supports, through function $tas()$, a randomized wait-free atomic test-and-set operation. The $tas()$ function, when invoked on element $W[i][j]$ of the matrix, tests if $W[i][j]$ is 0 and sets $W[i][j]$ to 1. If element $W[i][j]$ is 0, function $W[i][j].tas()$ returns $TRUE$ and we say that the process p that called $W[i][j].tas()$ *wins* or *succeeds* in the test-and-set operation. If element $W[i][j]$ is 1, function $W[i][j].tas()$ returns *FALSE* and we say that the process p that called $W[i][j].tas()$ loses the test-and-set operation. There are various randomized implementations for test-and-set on asynchronous shared memory. We use the *RatRace* algorithm from Alistarh *et. al.* [3].

Each of the n elements of the rows $\{1,\ldots,\log n\}$ of matrix W corresponds to one at-most-once job. Moreover, each of the $\frac{n}{\log n}$ elements of row 0 of matrix W is associated

RA **for process** p**:**

```
1.  FREE ← {0,...,n/log n − 1}
2.  size ← n/log n
3.  while(size > 0)
4.    next ← FREE.random()
5.    for(i ← 1, i ≤ log n, i++)
6.      if W[next][i].tas() then
7.        j = next · log n + i
8.        do_{j,p}
9.      endif
10.   endfor
11.   flag ← W[next][0].tas()
12.   if(flag) then
13.     head ← next
14.     tail ← next
15.     while(flag)
16.       W[next][0].setHead(head)
17.       W[head][0].setTail(tail)
18.       next++
19.       if(next < n/log n)then
20.         tail ← next
21.         for(i ← 1, i ≤ log n, i++)
22.           if W[next][i].tas() then
23.             j = next · log n + i
24.             do_{j,p}
25.           endif
26.         endfor
27.         flag ← W[next][0].tas()
28.       else
29.         flag ← FALSE
30.       endif
31.     endwhile
32.     size ← FREE.remove(head,tail)
33.   else
34.     head ← next
35.     tail ← next
36.     tmp ← W[next][0].getHead()
37.     if(tmp != ⊥)&&(tmp < head) then
38.       head ← tmp
39.     endif
40.     tmp ← W[head][0].getTail()
41.     if(tmp != ⊥)&&(tmp > tail) then
42.       tail ← tmp
43.     endif
44.     size ← FREE.remove(head,tail)
45.   endif
46. endwhile
```

Figure 7: Algorithm RA: pseudocode

with a super-job of $\log n$ jobs. In addition, each element of row 0 has two pointers: *head* and *tail*. The pointers are initially set to $\perp$. When their value is different from $\perp$ they point to elements of row 0. These elements correspond to the beginning and the end (respectively) of an interval of super-jobs that some process p has been working on. A process may access pointers *head* and *tail* through the $getHead()$, $getHead()$, $setTail()$ and $getTail()$ functions. Note that in RA an element of row 0 of matrix W is only set to 1

through an invocation of the test-and-set function associated with it. Moreover, if $W[i][0]$ is set to 1 for some $i \in \{0, \ldots, \frac{n}{\log n} - 1\}$, then for all $j \in \{1, \ldots, \log n\}$ we have that $W[i][j]$ is set to 1 through an invocation of the $W[i][j].tas()$ test-and-set function.

Internal Variables of process p. The variable FREE keeps the set of super-jobs that process p has not verified as performed. The variable FREE is a tree structure that keeps track of intervals of available super-jobs. Each interval of available jobs is stored in a leaf node as two pointers (beginning and ending of the interval). Initially the FREE set has only 1 leaf node that contains two pointers to 0 and $\frac{n}{\log n} - 1$. Process p interacts with FREE using 2 functions. The function $random()$ returns an element from the FREE set uniformly at random. Since the set FREE is stored in a tree structure, retrieval of a random element can be done in $O(l)$ where l the number of leaves of the tree. The function $remove(head, tail)$ removes from the set FREE the interval of elements beginning in *head* and ending in $tail$, or the subset of elements of the interval $\{head, \ldots, tail\}$ that intersects with the set FREE. The function returns as output the number of elements left in set FREE.

The variable *size* stores the number of elements in the set FREE. It is initially set to $\frac{n}{\log n}$ and it is only updated when the function $remove()$ is called on variable FREE.

The variables *head* and $tail$ hold the current working interval of super-jobs of process p or the interval of super-jobs that process p has learned to have been performed.

The variable *next* holds the next super-job process p is performing.

The variable tmp holds values of pointers fetched from the shared memory through the $getHead()$ or $getTail()$ functions.

The variable $flag$ normally holds the output of the latest test-and-set function called on a location of the 0 row and is used for exiting the inner *while* loop.

Finally variable i is used as index in for loops and j as index for at-most-once jobs,

in the do action.

Description of algorithm RA **for process** p**.** Initially process p sets the FREE set to contain all super-jobs and sets the *size* variable to $\frac{n}{\log n}$ which is the number of super-jobs. As long as there are more super-jobs to perform ($size > 0$) process p executes the following:

Process p picks a super-job to perform uniformly at random from set FREE (line 4). For each job grouped under the selected super-job, p calls the job's test-and-set operation and performs the job if it wins the associated test-and-set operation. Independently of whether the test-and-set operation was successful, p proceeds to the next job until all the test-and-set operations associated with the jobs under the super-job have been called (for loop in lines $5-10$). When all the test-and-set operations have been called, process p calls the test-and-set operation associated with the selected super-job (line 11). The procedure in lines $5-11$ is the same as the procedure in lines $21-27$. When process p completes this procedure we say that process p performed super-job *next*.

If process p wins the test-and-set operation on the randomly selected super-job, process p starts working on an interval of super-jobs. The interval starts at the randomly selected super-job and moves to the right. Process p peforms one super-job at a time (lines $21-27$), moving rightwards. As long as process p wins the test-and-set operations associated with the super-jobs, p keeps expanding the interval from the right side ($while$ loop in lines $15-31$). The interval ends when the first super-job test-and-set operation fails. When this happens, process p removes the interval from the FREE set (line 32) and as long as there are still super-jobs left in set FREE, picks a new random starting point and repeats the process. Note also that process p keeps the shared memory updated with the interval it is currently working at (or has just finished working), through the *head* and *tail* pointers of the row 0 of the W matrix (lines $16, 17$).

If the test-and-set operation on the randomly selected super-job fails, process p reads the *head* pointer of the respective shared memory location (line 36) and if it contains a valid value, it reads the *tail* pointer from the $W[head][0]$ location of the shared memory (line 40). Then it removes from the FREE set, the interval indicated by the *head* and *tail* pointers, and picks a new super-job at random. Essentially process p detects that it collided in the specific super-job with another process and attempts to increase the knowledge it has about which jobs have been performed, by learning the interval of super-jobs that have been completed, around the position of collision.

7.2 Analysis of Algorithm RA

We are going to prove that algorithm RA solves the strong at-most-once problem, by proving that it solves the at-most-once problem (no job is executed more than once), it has effectiveness $n - f_k$ and it is wait-free.

Then we will proceed with the work complexity analysis. We will prove that the expected work complexity of the algorithm is $O(n + k^{2+\varepsilon} \log n)$, for any constant ε. We will do that by proving that in order to perform the first $n - k^2 \log n$ jobs, the expected work of the algorithm is $O(n)$. Then we will prove that RA will need $O(k^{2+\varepsilon} \log n)$ work to perform the next $k^2 \log n - k^{1+\varepsilon} \log n$ jobs. Finally RA will need no more than $O(k^{2+\varepsilon} \log n)$ work for the remaining jobs.

In order to show that algorithm RA solves the strong at-most-once problem, we prove that RA solves the at-most-once problem (no job is executed more than once), it has strong optimal effectiveness and is wait-free. The first two parts are straight forward. Based on the correctness properties of the test-and-set operations, it is easy to see that a job cannot be performed more than once. Moreover, in order to prove strong optimal effectiveness, we need to observe that in RA, a job will be performed unless some process

p wins the test-and-set operation associated with it, and then crashes before it performs the job. After this observation, we only need to show, that all test-and-set operations will be invoked by some process before RA terminates, and that after winning a test-and-set operation, a process will not call another test-and-set operation before it completes the job associated with the test-and-set it has already won. We still need to prove the wait-free property. The proof is based on the observation that every time the main loop is executed by process p, at least on element will be removed from the local FREE set of p. The set has finite elements and thus process p will have to terminate if it does not fail. Following the strategy above we can prove the Theorem 7.4.

Lemma 7.1. *There exists no execution α of algorithm* RA*, such that $\exists i \in \mathcal{J}$ and $\exists p, q \in \mathcal{P}$ for which $\mathsf{do}_{p,i}, \mathsf{do}_{q,i} \in \alpha$.*

Proof. If such an execution α exists, it means that processes p, q must succeed in the call of the same test-and-set operator. This is impossible from the properties of the test-and-set operator. This holds even if $p = q$ since p will invoke an $\mathsf{do}_{p,i}$ operation only once after succeeding in the respective test-and-set operator. □

Lemma 7.2. *There exists no infinite execution of algorithm* RA.

Proof. Let us assume that there exists infinite execution α of algorithm RA. Since execution α is infinite, there exists at least one process p in α that has not crashed, has not terminated and takes infinite steps in execution α. There exist 2 while loops in algorithm RA. One in lines $4-47$ and one in $16-32$.

The second while loop cannot be infinite since as long as the condition in line 16 is $TRUE$, in each iteration *next* is increased (line 19) and it will eventually become greater than $\frac{n}{\log n}$. This will make the if statement in line 20 false, and as a result the condition in line 16 will be set to *FALSE*.

Thus the first while loop should be infinite. This means that the check in line 4 of the algorithm will remain $TRUE$ for process p and thus p will always have at least 1 element inside its FREE set. In each iteration of the while loop (line $4-47$), p picks at random a new element from set FREE and removes at least this element from set FREE, either in line 33 or line 45. This holds because if in line 12 the if statement is true, then *head* is set to the selected element and *tail* to something that is equal (line 14) or greater (line 21) to the selected element. If in line 12 the if statement is false, we have (lines $35-43$) that either the selected element or the selected element with some more elements is removed from the set FREE.

In each iteration of the loop of lines $4-47$ at least 1 element is removed from set FREE, FREE has a finite number of elements and no elements are added in set FREE in algorithm RA. It must be the case that the FREE set will become empty after a finite number of iterations and thus process p will have to terminate. □

Lemma 7.3. *Algorithm* RA *has effectiveness* $E_A(n,m,f,f_k) = n - f_k$

Proof. We are first going to show that if a process p terminates, then all the test-and-set operations have been called by at least one process. Then we will show that no jobs are lost to processes that do not participate in an execution. Moreover we will demonstrate that a process that fails after taking steps in an execution can cost algorithm RA at most one job, thus leading to a effectiveness of $E_A(n,,m,f,f_k) = n - f_k$.

If a process p terminates it means that the condition in the while loop of lines $4-47$ is false and the FREE set is empty. Process p removes elements from the FREE set in lines 33 and 45. When process p removes elements from *head* to *tail* in line 33, we have that process p has called all the $(tail - head + 1) \cdot \log n$ test-and-set operations associated with the jobs of the $(tail - head + 1)$ super-jobs that correspond to the *head* to *tail* elements of the FREE set. Moreover process p has won all the $(tail - head)$ test-

and-set operations that are associated with the super-jobs from *head* to *tail*, unless the end of the W array has been met, in which case $(tail - head) + 1$ test-and-set operations for super-jobs have been won. This means that all the test-and-set operations associated with the jobs from $head \cdot \log n + 1$ to $tails \cdot \log n + \log n$ have been called by process p. Moreover, the $W[head].[0].tail$ variable in the shared memory has value $tail - 1$, unless the end of the W array was met in which case it contains the value $tail$. Also for all $i \in \{head, \ldots, tail - 1\}$, $W[i].[0].head$ has value $head$. If the end of the W array was met, then $W[tail].[0].head$ also has value $head$.

When process p removes elements from *head* to *tail* in line 45, we have 2 cases.

Case 1: Both if statements in lines 38 and 42 returned false. This means that in line 45, only the element *next* is removed. From lines $6-10$ we have that all the test-and-set operations associated with the jobs of the super-job *next* have been called by p.

Case 2: At least one if statement in lines 38 and 42 returned true. Then there exists a process q that has won the test-and-set operations for super-jobs *head* and $tail$. Moreover for Fig. 7 we have that process q has called all the test-and-set operations associated with the jobs from $head \cdot \log n + 1$ to $tails \cdot \log n + \log n$.

From the discussion above we have that if a process p terminates, all the test-and-set operations have been called by at least one process.

We continue with the observation that only a process that calls a test-and-set operation may succeed in it. This means that processes that do not call a test-and-set operation cannot win this test-and-set operation. Thus processes that do not participate in an execution, cannot win test-and-set operations in it. Moreover from Fig. 7 we have that when a process succeeds in a test-and-set operation that is associated with a job i (lines $6-11$ and $22-27$) , then the process performs that job, before calling a new test-and-set operation. This means that at any given state in an execution α, a process p may have won at most

one more test-and-set operation (associated with jobs), than the jobs it has performed. If the process fails at this point, the job associated with the specific test-and-set operation will never be performed. Moreover as we discussed above a process p must have called the specific test-and-set operation in order to win it, and thus must have taken at least one step in execution α.

From Lemma 7.2 we have that there exist no infinite executions for algorithm RA. This means that all fair executions are finite and thus all non-failed processes terminate in a fair execution. From the discussion above, since all processes terminate in a fair execution, all the test-and-set operations associated with jobs are called by at least one process. Processes that do not participate in the execution cannot win test-and-set operations and each process that participates in an execution and crashes, may result in at most one job being lost. All processes that do not crash and win a test-and-set operation, will eventually perform the job associated with it. Thus we have that the effectiveness of algorithm RA is $E_A(n,m,f,f_k) = n - f_k$. □

Theorem 7.4. *Algorithm* RA *is wait-free and solves the strong at-most-once problem with optimal effectiveness* $n - f_k$.

Proof. Theorem 7.4 follows directly from Lemmas 7.1, 7.2 and 7.3. □

In order to prove that the expected work complexity of the algorithm is $O(n + k^{2+\varepsilon} \log n)$, for any constant ε, we use the following strategy. We observe that at any point of the execution, the set of available super-jobs is split in at most $k+1$ intervals. Using this observation we can prove that when a process samples a new super-job i in order to start a new interval, if i belongs in the set of available super-jobs, it belongs in one of the $k+1$ intervals, and the expected distance of the super-job from the ends of the intervals is large enough, to allow significant progress, before the process need to sample again. Using the same strategy, we can also show that when a process stops working

in an interval of super-jobs the expected size of the completed interval it leaves behind, is significant. Based on the above, we prove that whenever the process samples an element, it will either be able to perform significant work before it need to sample again, or it will update its knowledge about performed super-jobs with significant information. Using the above, we first show that for performing $n-k^2$ super-jobs and learning that those $super-jobs$ have been performed, algorithm RA needs excepted work $O(n)$. At this point k^2 super-jobs still need to be performed. Next we show that for any constant ε, in order to perform the next $k^2-k^{1+\varepsilon}$ super-jobs RA needs excepted work $O(k^{1+\varepsilon}\log n)$. Finally for the last $k^{1+\varepsilon}$ super-jobs it needs again excepted work $O(k^{1+\varepsilon}\log n)$. With the above strategy, we get the expected work complexity in Theorem 7.12.

For a state s of an execution α of algorithm KK_β, we define as $FREE_s$ the set of super-jobs whose associated test-and-set operations have not been invoked by any process. We say that these super-jobs are ***available*** at state s. We denote by $FREE_{s,p}$, the value of the internal variable FREE of process p in state s.

Lemma 7.5. *For all executions of algorithm* RA *where k processes participate, the available super-jobs are split into no more than* $k+1$ *intervals.*

Proof. The k processes that participate in algorithm RA initially create at most k working intervals of super-jobs and thus, there at most $k+1$ intervals of *available* super-jobs. Each process p stops working on its current interval at some super-job i if it loses a super-job test-and-set operation on i (i now denotes the end of the interval), or if $i=\frac{n}{\log n}-1$ (the end of array W). We will show that in both cases, the number of working intervals does not increase or equivalently the number of intervals of *available* super-jobs does not increase.

If process p loses a test-and-set operation at super-job $i<\frac{n}{\log n}-1$, it must be the case that some other process q has won the test-and-set operation at i and is working (or was working) on an interval starting at position i. The two intervals are merged and although

process p might create a new working interval, the number of working intervals does not increase.

If process p reaches the end of array W, it must be the case that the interval of *available* super-jobs at the end of W does not exist anymore and although process p has to create a new working interval, the number of *available* super-jobs intervals does not increase.

From the two observations above, we have that at most $k+1$ intervals of *available* super-jobs can exist for any state s in any execution α of algorithm RA. □

At any state of an execution α of algorithm RA, we have that $\text{FREE}_s \subseteq \text{FREE}_{s,p}$. This holds since FREE_s is the set of the available super-jobs at state s, while is $\text{FREE}_{s,p}$ the estimation that process p has on which super-jobs are still available. From Lemma 7.5, we have that FREE_s is separated in at most $k+1$ intervals. Let these intervals be $S_0,\ldots,S_k$, such that for $i \in \{0,\ldots,k\}$, $S_i = [a_i, b_i]$ where a_i, b_i the first and last element of S_i, or $S_i = \emptyset$. Moreover, it holds that $\text{FREE}_s = \bigcup_{i=0}^{k} S_i$ and $\sum_{i=0}^{k} |S_i| = |\text{FREE}_s|$ since the intervals S_i are disjoint. Let $d_i = |S_i|$ for $i \in \{0,\ldots,k\}$.

If process p in line 4 of algorithm RA, gets *next* where $next \in \text{FREE}_s$, then $\exists i \in \{0,\ldots,k\}$ such that $next \in S_i$. Let X be the random variable that represents the minimum distance of *next* from the two endpoints of interval S_i, namely $X = \min(next - a_i, b_i - next)$. We are going to show that $E[X] \geq \frac{1}{2}\left(\frac{|\text{FREE}_s|}{k+1} - 1\right)$.

Lemma 7.6. *If at state s of execution α process p samples an elements next from the set* FREE_s*, we have that* $E[X] \geq \frac{1}{2}\left(\frac{|\text{FREE}_s|}{k+1} - 1\right)$

Proof. For the expected value of X we have:

$$
\begin{aligned}
E[X] &= \sum_{i=0}^{k} \left(\frac{d_i}{|\text{FREE}_s|} \cdot \sum_{j=a_i}^{b_i} \left(\frac{min(j-a_i, b_i-j)}{d_i} \right) \right) && (8) \\
&= \frac{1}{|\text{FREE}_s|} \sum_{i=0}^{k} \sum_{j=0}^{d_i} min(j, d_i) && (9) \\
&\geq \frac{1}{|\text{FREE}_s|} \sum_{i=0}^{k} 2 \sum_{j=0}^{\frac{d_i}{2}-1} j && (10) \\
&= \frac{1}{|\text{FREE}_s|} \sum_{i=0}^{k} 2\frac{d_i}{4} \left(\frac{d_i}{2} - 1 \right) && (11) \\
&\geq \frac{1}{|\text{FREE}_s|} \sum_{i=0}^{k} \frac{1}{2} \left(\frac{d_i^2}{2} - d_i \right) && (12)
\end{aligned}
$$

We have that $|\text{FREE}_s| = \sum_{i=0}^{k} d_i$, so it holds that

$$
\sum_{i=0}^{k} d_i^2 \geq \sum_{i=0}^{k} \left(\frac{|\text{FREE}_s|}{k+1} \right)^2
$$

So from 12 we have:

$$
E[X] \geq \frac{1}{|\text{FREE}_s|} \sum_{i=0}^{k} \frac{1}{2} \left(\left(\frac{|\text{FREE}_s|}{k+1} \right)^2 - d_i \right) \Rightarrow E[X] \geq \frac{1}{2} \left(\frac{|\text{FREE}_s|}{k+1} - 1 \right)
$$

□

Next we are going to show that if $|\text{FREE}_s| \geq k^2$, then when a process p starts working in an interval of super-jobs, we expect it to be able to perform $\Omega(k)$ super-jobs, before it loses a super-job test-and-set operation to some other process.

Lemma 7.7. *If for all states s of execution α, $|\text{FREE}_s| \geq k^2$, then if a process p loses a super-job test-and-set operation by executing line 27 in execution α, then the expected size of p's working interval is greater than $0.33(k-1)$.*

Proof. Since process p passed line 12 in the algorithm RA, in the state s when p executed line 04, process p sampled an element $i \in \text{FREE}_s$ set. From Lemma 7.6, we have that i belongs in an interval of available super-jobs and that the expected distance of i from the endpoints of the interval is greater than $\frac{1}{2}\left(\frac{|FREE_s|}{k+1} - 1\right) \geq \frac{1}{2}\left(\frac{k^2}{k+1} - 1\right)$. The later is greater than $0.33(k-1)$ for any $k > 1$. Since in line 11 process p invokes the test-and-set operation for super-job i, any process that succeeds in sampling an available super-job in execution α, after p executed line 11, again from Lemma 7.6 will have an expected distance from i greater than $0.33(k-1)$.

From the observations aboves, it follows that if p loses a test-and-set operation on super-job j, while working on the interval that begins in i, the expected distance between i and j is $0.33(k-1)$. □

Let states s, s' be the states of two consecutive executions of line 4 by process p in execution α (s precedes s' in α). We define the random variable Y as $|\text{FREE}_{s,p}| - |\text{FREE}_{s',p}|$. Observe that Y is the number of super-jobs a process p was able to remove from its FREE set, as a result of randomly selecting *next* from the set $\text{FREE}_{s,p}$ in state s. There are three cases for the sampled element *next*: a) *next* can be in set FREE_s, of the available super-jobs in state s (event B), b) it can be in an interval of performed super-jobs that some process q has performed and has stopped expanding because it lost super-job test-and-set operation by executing line 27 in execution α (event C), or c) it can be in an interval of performed super-jobs, that some process q has performed, but process q has not lost a super-job test-and-set operation by executing line 27 yet (event D). This means that process q has either failed or is still expanding the interval. Observe that in event D the adversary has complete control over the size of the $(k-1)$ intervals in state s. On the other hand Lemma 7.6 applies for events B and C.

We examine first the expected value of Y given event D. There can be at most $k-1$

intervals where the $k-1$ processes (all process but p) are currently working, or failed. Let the size of those intervals be d_i for $i \in [1,k-1]$, we have $E[Y|D] = d = \frac{1}{k-1}\sum_{i=1}^{k-1} d_i$ and $E[Y] = \frac{|\text{FREE}_s|}{|\text{FREE}_{s,p}|}E[Y|B] + \frac{|\text{FREE}_s p| - |\text{FREE}_s| - (k-1)d}{|\text{FREE}_{s,p}|}E[Y|C] + \frac{(k-1)d}{|\text{FREE}_{s,p}|}E[Y|D]$

Lemma 7.8. *$E[Y] = \Omega(k)$, if for all states s of an execution α $|\text{FREE}_s| \geq k^2$.*

Proof. We have that:

$$E[Y] = \frac{|\text{FREE}_s|}{|\text{FREE}_{s,p}|}E[Y|B] + \frac{|\text{FREE}_{s,p}| - |\text{FREE}_s| - (k-1)d}{|\text{FREE}_s p|}E[Y|C] + \frac{(k-1)d}{|\text{FREE}_{s,p}|}E[Y|D]$$

We have that $E[Y|B] \geq 0.33(k-1)$ and $E[Y|C] \geq 0.33(k-1)$ from Lemmas 7.6 and 7.7, since for any state s of execution α $|\text{FREE}_s| \geq k^2$.

Thus we get:

$$E[Y] \geq \frac{|\text{FREE}_s|}{|\text{FREE}_{s,p}|}0.33(k-1) + \frac{|\text{FREE}_{s,p}| - |\text{FREE}_s| - (k-1)d}{|\text{FREE}_{s,p}|}0.33(k-1) + \frac{(k-1)d}{|\text{FREE}_{s,p}|}d$$

If $d \geq k-1$ we have that $E[Y] = \Omega(k)$. If $(k-1) > d$, since $|\text{FREE}_s| \geq k^2$ we have:

$$\begin{aligned} E[Y] &\geq \frac{k^2}{k^2+(k-1)d}0.33(k-1) + \frac{(k-1)d}{k^2+(k-1)d}d \\ &= (k-1)\frac{0.33k^2+d^2}{k^2+(k-1)d} \\ &= (k-1)\frac{0.33+\frac{d^2}{k^2}}{1+\frac{(k-1)d}{k^2}} \end{aligned}$$

Since $k > d$ there exists constant c such that $\frac{0.33+\frac{d^2}{k^2}}{1+\frac{(k-1)d}{k^2}} < c$, from which we get that $E[Y] = \Omega(k)$. □

Lemma 7.9. *In order to perform $n-k^2$ super-jobs,* RA *needs expected work* $\text{O}(n)$.

Proof. From Lemma 7.8, we have that the expected number of times a process p executes line 4 of algorithm RA is $O(\frac{n}{k \log n})$. If a process p loses the test-and-set in position *next* (line 11), we have $O(\log n)$ work for the for loop in lines $5-10$. Additionally, we have $O(1)$ work for lines $34-43$ and $O(\log n)$ for the removal of interval $[head, tail]$ from set FREE in line 44.

This holds because the set FREE is a tree structure with intervals of available super-jobs as leafs, as a result there can be no more than $\frac{n}{\log n}$ leafs in tree FREE. Moreover there exists one process q that has won all the test-and-set operations on the interval $[head, tail]$ and super-jobs in $[head, tail]$ have been performed by process q as the result of a single execution of line 4. So either process p has all the super-jobs of interval $[head, tail]$ in its FREE set, or there exists super-job i in interval $[head, tail]$, such that all super-jobs in the interval $[i+1, tail]$ are in the FREE set of process p and the super-jobs in the interval $[head, i]$ are not in the FREE set. This can happen if process p sampled an element in the interval $[head, i]$, while process q was performing the interval $[head, tail]$, but it had only updated the related information in array W for the super-jobs from *head* to i.

In the first case, there exists one leaf node in FREE that contains interval $[i, j]$ of available super-jobs, such that $i \leq head$ and $j \geq tail$. If $i = head$ and $j = tail$, the removal of interval $[head, tail]$ from FREE will result in the removal of the leaf node. If $i < head$ and $j > tail$ the removal of interval $[head, tail]$ from FREE will result in the creation of a new leaf node. Finally, if $i = head$ and $j > tail$ or $i < head$ and $j = tail$ the removal of interval $[head, tail]$ from FREE will result in the update of one end of the current node. Each of those operations has work $O(\log n)$, since the tree has less than $\frac{n}{\log n}$ leafs.

In the second case, there exists one leaf node in FREE that contains interval $[i, j]$ of available super-jobs, such that $i > head$ and $j \geq tail$. Moreover the super-jobs in the interval $[head, i)$ are not in FREE. If $j = tail$ the removal of interval $[head, tail]$ from

FREE will result in the removal of the leaf node. If $j > tail$ the removal of interval $[head, tail]$ from FREE will result in the update of the left end of the leaf node so that it now contains $[tail+1, j]$. Again each of those operations has work $O(\log n)$, since the tree has less than $\frac{n}{\log n}$ leafs.

If a process p wins the test-and-set in position *next* (line 11), we have $O(\log n)$ work for the for loop in lines $5-10$. Additionally every super-job performed in lines $19-27$ adds $O(\log n)$ work plus $O(1)$ work for updating the shared memory lines $16, 17$. Finally with similar arguments as before we can show that the removal of interval $[head, tail]$ from the set FREE in line 32, costs $O(\log n)$.

From the above discussion, for each of the k processes, we have $O(\frac{n}{k})$ work when the processes lose the test-and-set operation in line 11. Moreover, when processes win line 11 we have for all processes a total of $O(n)$ work for performing the $n-k^2$ super-jobs. The above add up to $O(n)$ work. We have not discussed the cost of the randomized test-and-set operations that we are using. As mentioned above we use the *RatRace* algorithm from Dan Alistarh *et. al.* [3], in order to implement the randomized test-and-set operations. The algorithm is wait-free and k-adaptive and has expected work $O(k \log k)$, where k the jobs participating in the a specific test-and-set. It is easy to see that the expected number of processes participating in any specific test-and-set operation is $O(1)$, thus the expected work from each test-and-set operation in $O(1)$. Thus the work of RA for performing $n-k^2$ super-jobs is $O(n)$.

□

Next we will analyze the work needed by algorithm RA in order to perform the remaining k^2 super-jobs. Start by generalizing Lemma 7.8.

Lemma 7.10. *If for all states s of an execution* α $|\text{FREE}_s| \geq k^{2-i\varepsilon}$, *we have that* $E[Y] = \Omega(k^{1-i\varepsilon})$ *for some small constant* ε.

Proof. We have that:

$$E[Y] = \frac{|\text{FREE}_s|}{|\text{FREE}_{s,p}|}E[Y|B] + \frac{|\text{FREE}_{s,p}| - |\text{FREE}_s| - (k-1)d}{|\text{FREE}_s p|}E[Y|C] + \frac{(k-1)d}{|\text{FREE}_{s,p}|}E[Y|D]$$

We have that $E[Y|B] \geq 0.33(k^{1-i\varepsilon} - 1)$ and $E[Y|C] \geq 0.33(k^{1-i\varepsilon} - 1)$ from Lemmas 7.6, since for any state s of execution α $|\text{FREE}_s| \geq k^{2-i\varepsilon}$.

Thus we get:

$$E[Y] \geq \frac{|\text{FREE}_s|}{|\text{FREE}_{s,p}|}0.33(k^{1-i\varepsilon} - 1) + \frac{|\text{FREE}_{s,p}| - |\text{FREE}_s| - (k-1)d}{|\text{FREE}_{s,p}|}0.33(k^{1-i\varepsilon} - 1) + \frac{(k-1)d}{|\text{FREE}_{s,p}|}d$$

If $d \geq k^{1-i\varepsilon} - 1$ we have that $E[Y] = \Omega(k^{1-i\varepsilon})$. If $(k^{1-i\varepsilon} - 1) > d$, since $|\text{FREE}_s| \geq k^{2-i\varepsilon}$ we have:

$$\begin{aligned} E[Y] &\geq \frac{k^{2-i\varepsilon}}{k^{2-i\varepsilon} + (k-1)d}0.33(k^{1-i\varepsilon} - 1) + \frac{(k-1)d}{k^{2-i\varepsilon} + (k-1)d}d \\ &= (k^{1-i\varepsilon} - 1)\frac{0.33k^{2-i\varepsilon} + \frac{(k-1)d^2}{k^{1-i\varepsilon}-1}}{k^{2-i\varepsilon} + (k-1)d} \\ &= (k^{1-i\varepsilon} - 1)\frac{0.33 + \frac{(k-1)d^2}{k^{2-i\varepsilon}(k^{1-i\varepsilon}-1)}}{1 + \frac{(k-1)d}{k^{2-i\varepsilon}}} \end{aligned}$$

Since $k^{1-i\varepsilon} - 1 > d$ there exists constant c such that $\frac{0.33 + \frac{(k-1)d^2}{k^{2-i\varepsilon}(k^{1-i\varepsilon}-1)}}{1 + \frac{(k-1)d}{k^{2-i\varepsilon}}} < c$, from which we get that $E[Y] = \Omega(k^{1-i\varepsilon})$. □

Next we analyze the work needed by algorithm RA in order to perform $k^2 - k^{1+\varepsilon}$ super-jobs for some small constant ε, given that initially there exists k^2 available super-jobs. We show this by induction using Lemma 7.10.

Lemma 7.11. *Algorithm* RA *needs expected work* $O(k^{2+\varepsilon}\log n)$ *in order to perform* $k^2 - k^{1+\varepsilon}$ *super-jobs, for some small constant* ε*, given that only* k^2 *super-jobs are available and processes know that only the specific* k^2 *super-jobs are available.*

Proof. (proof for Lemma 7.11)

Base Case: From Lemma 7.10, for $i = 1$ we have that the expected number of times a process p executes line 4 of algorithm RA, till $k^2 - k^{2-\varepsilon}$ super-jobs are performed and process p updates its FREE set to remove these super-jobs is $O(k^{1+\varepsilon})$. This holds because $E[Y] = \Omega(k^{1-\varepsilon})$ and initially all process FREE sets have k^2 super-jobs. With similar arguments as in Lemma 7.9 we have that algorithm RA needs work $O(k^{2+\varepsilon}\log n)$.

Iterative Step: We are going to find the work algorithm RA needs for some i in order to perform $k^{2-i\varepsilon} - k^{2-(i+1)\varepsilon}$ super-jobs given that only $k^{2-i\varepsilon}$ super-jobs are available and processes start knowing that only the specific $k^{2-i\varepsilon}$ super-jobs are available. From Lemma 7.10, we have that the expected number of times a process p executes line 4 of algorithm RA, till $k^{2-i\varepsilon} - k^{2-(i+1)\varepsilon}$ super-jobs are performed and process p updates its FREE set to remove these super-jobs is $O(k^{1+\varepsilon})$. This holds because $E[Y] = \Omega(k^{1-(i+1)\varepsilon})$ and initially all process FREE sets have $k^{2-i\varepsilon}$ super-jobs. With similar arguments as in Lemma 7.9 we have that algorithm RA needs work $O(k^{2+\varepsilon}\log n)$.

We need $\frac{1}{\varepsilon} - 2$ steps till $2 - (i+1)\varepsilon = 1 + \varepsilon$. This is a constant and thus the total work needed is $O(k^{2+\varepsilon}\log n)$. □

Theorem 7.12. *Algorithm* RA *has expected work complexity* $O(n + k^{2+\varepsilon}\log n)$ *for any small constant* ε*.*

Proof. From Lemmas 7.9 and 7.11 we have that in order to perform $n - k^{1+\varepsilon}$ super-jobs and for the processes to learn that those super-jobshave been performed, RA needs work $O(n + k^{2+\varepsilon}\log n)$. If there are $\Omega(k)$ available super-jobs, we expect that any test-end-end operation will have $O(1)$ processes calling it. If there are $O(k)$ available super-jobs,

the we expect that test-end-end operations will contribute no more that a multiplicative $O(\log k)$ factor in the algorithm. From the above discussion we have the performing the remaining $k^{1+\varepsilon}$ super-jobs will contribute $O(k^{2+\varepsilon}\log n + k^2 \log n \log k) = O(k^{2+\varepsilon}\log n)$ expected work. Thus we have that for any small constant ε algorithm RA has expected work complexity $O(n + k^{2+\varepsilon}\log n)$. □

7.3 Open Problems

One question that arises from Theorem 3.7, is what kind of deterministic solutions can we expect for the strong at-most-once problem. Clearly such solutions cannot be wait free. It is interesting to study what progress requirements are reasonable for such solutions. Partial synchrony or failure detectors could be possible ways to circumvent the impossibility result.

It would be interesting to study a lower bound on work for randomized algorithms, as well as a lower bound on work for k-adaptive algorithms both in the context of the at-most-once problem, as well as the write-all. The lower bound of $\Omega(n + p\log n)$ from [43] is a good indication, but how does this change for k-adaptive algorithms? Can solutions of $\Omega(n + k\log n)$ be achieved?

8 Randomized Adaptive Write-All Algorithm ARTA

In this section we present and analyze an asynchronous randomized algorithm called ARTA that solves the Write-All problem. The algorithm is presented using pseudocode. We use the adaptive adversary presented in Section 2. Our memory supports atomic read/write registers and atomic test-and-set operations. We show that ARTA has high probability work complexity $O(n + k^2 \log^3 k)$.

8.1 Algorithm ARTA

In algorithm ARTA (Figures 8, 9, 10) each task is associated with a shared memory location of vector W. The vector W has n elements. Each element of W supports a wait-free atomic test-and-set operation and two pointers named head and tail that point to positions of vector W. The pointers are initialized to the value "$\perp$". The test-and-set operations are used in order to detect and resolve collisions between processes. The pointers are used in order to mark intervals of completed tasks. This is important since processes learn about the completion of tasks in terms of these intervals.

ARTA **for process** p**:**

1. FREE $\leftarrow [0, n-1]$
2. DONE $\leftarrow \emptyset$
3. $size \leftarrow n$
4. **while**$(size > 0)$
5. $\quad$ [FREE, DONE, $size$] $\leftarrow$ *progress*(FREE, DONE, $size$)
6. $\quad$ [FREE, DONE, $size$] $\leftarrow$ *defrag*(FREE, DONE, $size$)
7. **endwhille**

Figure 8: Algorithm ARTA: driver script

Below we will give an overview of algorithm ARTA, highlighting the key ideas and their impact on work complexity. Processes in algorithm ARTA operate in two "steps", a progress step (Fig. 9) and a defragmentation step (Fig. 10). During the progress step, a process creates an interval of completed tasks or discovers an interval of completed tasks another process created. During the defragmentation step, a process revisits an interval of completed tasks it knows, and examines whether it can be expanded and merged with other intervals it knows. This can happen if all the tasks between the two intervals have been performed.

In the progress step, the main idea is that every process p picks a random task i that the process believes is available, as a candidate starting point for its interval. If process p wins the test-and-set operation for task i, it marks i as the starting point of its working interval, and moves to the next task in a rightward direction, one task at a time, until it loses a test-and-set operation. As long as p wins test-and-set operations, it adds the tasks to its current interval. If it fails this means that some other process has marked the specific task as the beginning of its working interval. In this case, p stops working on the interval. If process p loses the first test-and-set operation (for task i), it learns the interval of completed tasks in which task i belongs by reading the respective pointers. In this way, process p learns about progress that some other process did.

The key idea behind this approach is that a process p keeps expanding a working interval that started at some task i until it loses a test-and-set operation at some task j on the right of task i. This means that some other process q won that test-and-set for j and thus q continues expanding the interval that started at i. This leads to the observation that if k processes are participating in an execution of the algorithm ARTA, there exist at most k intervals of performed tasks, from which, at any given point of the execution, at most k are being expanded from the right end.

The observation that there are at most k expanding intervals, helps us form probabilistic arguments on the fragmentation of the memory and how much progress can be done before a process loses a test-and-set operation. Clearly the k intervals of performed tasks, are composed from smaller intervals that have been preformed by distinct processes. A process spends constant time in order to learn each of these smaller intervals. By showing that the number of these smaller intervals is bounded by $O(\frac{n}{k \log k} + k \log^2 k)$, we will show that the cost for learning in our algorithms is kept low.

Intuitively, as long as the available tasks are significantly more than the k processes

participating in an execution of the algorithm, processes that need a new random starting point, will likely be positioned far enough from the endpoints of the existing intervals of performed tasks. This results in substantial progress being done before a process working on an interval loses a test-and-set operation and thus has to start a new interval. The latter implies that if a process has an outdated estimation of the available tasks, by colliding with large intervals of performed tasks, it will be able to learn fast about tasks that have been completed.

The processes learn about performed tasks in two ways. In the progress step, if a process loses the first test-and-set operation it learns about the interval of performed tasks that includes the task i which it tried to use as a starting point. A process can also learn about performed tasks during the defragmentation step. During that step, a process takes an interval of performed tasks and tries to expand it from the left side, by checking if other processes have completed tasks at the left of the interval. The process repeats this procedure until no more performed tasks remain at the left of the interval. This leftward expansion could lead in the merging in one bigger interval of multiple intervals of performed tasks the process knew about, thus reducing the fragmentation of the process' knowledge. This is of paramount importance for the work complexity of the algorithm, since we want to keep the knowledge of performed task fragmented in no more that $O(k)$ intervals, in order to perform operations with $O(\log k)$ work in the internal memory of the process.

Next we present the shared memory variables, internal variables and the steps a process p has to take in algorithm ARTA in more detail:

Shared Variables. Algorithm ARTA uses vector W of size n. The vector is stored in shared memory. Each element of the vector is initially set to 0 and supports, through function $tas()$, a wait-free test-and-set operation. The $tas()$ function, when invoked on

function *progress*(FREE, DONE, *size*) **for process** p**:**

```
1.  [next] ← FREE.randomElement()
2.  do_{p,next}
3.  win ← W[next].tas()
4.  head ← next
5.  tail ← next
6.  if(win) then
7.     while(win)
8.        W[tail].setHead(head)
9.        W[next].setTail(tail)
10.       next = (next + 1)  mod n
11.       head ← next
12.       do_{p,next}
13.       win ← W[next].tas()
14.    endwhile
15. else
16.    tmp ← W[next].getTail()
17.    if(tmp != ⊥) then
18.       tail ← tmp
19.    endif
20.    tmp ← W[tail].getHead()
21.    if(tmp != ⊥) then
22.       head ← tmp
23.    endif
24. endif
25. size ← FREE.remove(tail, head)
26. DONE.add(tail, head)
27. return [FREE, DONE, size]
```

Figure 9: Algorithm ARTA: function *progress*()

element $W[i]$ of the matrix, tests if $W[i]$ is 0 and sets $W[i]$ to 1. If element $W[i]$ is 0, function $W[i].tas()$ returns TRUE and we say that the process p that called $W[i].tas()$ *wins* or *succeeds* in the test-and-set operation. If element $W[i]$ is 1, function $W[i].tas()$ returns FALSE and we say that the process p that called $W[i].tas()$ loses the test-and-set operation.

Each of the n elements of vector W corresponds to one task. The tasks are mapped to the vector elements through their indexes (element $W[i]$ corresponds to task i). In addition, each element of W has two pointers: *head* and *tail*. The pointers are initially set to $\perp$. When their value is different from $\perp$ they point to elements of vector W. These elements correspond to the end and the beginning (respectively) of an interval of tasks that some process p has been working on. A process may access pointers *head* and *tail* through the *setHead*(), *getHead*(), *setTail*() and *getTail*() functions. Note that in ARTA an element

of vector W is only set to 1 through an invocation of the test-and-set function associated with it.

---

function *defrag*(FREE, DONE, *size*) **for process** p**:**

```
1.  [left, right] ← DONE.randomInterval()
2.  while(W[(left − 1) mod n])
3.     next ← (left − 1) mod n
4.     head ← next
5.     tail ← next
6.     tmp ← W[next].getTail()
7.     if(tmp != ⊥) then
8.        tail ← tmp
9.     endif
10.    tmp ← W[tail].getHead()
11.    if(tmp != ⊥) then
12.       head ← tmp
13.    endif
14.    size ← FREE.remove(tail, head)
15.    [left, right] ← DONE.add(tail, head)
16. endwhile
17. return [FREE, DONE, size]
```

Figure 10: Algorithm ARTA: function *defrag*()

Internal Variables of process p. The variable FREE keeps the set of task intervals that process p believes are available. The variable DONE keeps the set of task intervals that process p has verified as being performed. The variables FREE and DONE are tree structures that keep track of intervals of tasks. Each interval of tasks is stored in a leaf node as two pointers (beginning and ending of the interval). Initially the FREE set has only 1 leaf node that contains two pointers to 0 and n, while the DONE tree is empty. Process p interacts with FREE using 2 functions. The function *randomElement*() returns an element from the FREE set uniformly at random. Since the set FREE is stored in a tree structure, retrieval of a random element can be done in $O(\log l)$ where l is the number of leaves of the tree. The function *remove*(*tail*, *head*) removes from the set FREE the interval of elements beginning in *tail* and ending in *head*, or the subset of elements of the interval $\{tail, \ldots, head\}$ that intersects with the set FREE. The function returns as output the number of elements left in set FREE. Process p interacts with DONE using

2 functions. The function *randomInterval*() returns an interval from the DONE set uniformly at random. Specifically if DONE is composed from l intervals, each represented as a leaf node in the tree structure DONE set is stored at, function *randomInterval*() returns one of the leafs, picked with probability $\frac{1}{l}$. The function $add(tail, head)$ adds in the set DONE the interval of elements beginning in $tail$ and ending in $head$, or the subset of elements of the interval $\{tail, \ldots, head\}$ that do not belong in the set DONE. The function returns as output the interval $[left, right]$, which identifies the interval in the set DONE that now contains the interval $\{tail, \ldots, head\}$. We will show in 8.13 that the FREE and DONE sets have $O(k)$ leafs with high probability. This means that each call of the functions *randomElement*(), *randomInterval*(), $remove(tail, head)$ and $add(tail, head)$ will require $O(\log k)$ work with high probability. This happens because the functions $remove(tail, head)$ and $add(tail, head)$ interfere with at most two leaf nodes of the FREE and DONE set respectively.

The variable *size* stores the number of elements in the set FREE. It is initially set to n and it is only updated when the function *remove*() is called on variable FREE.

The variables *head* and $tail$ hold the endpoints of the current working interval of tasks of process p or the endpoints of an interval of tasks that process p has learned that it has been performed by some other process.

The variables $left$ and *right* hold the endpoints of the performed interval of tasks, that process p is trying to expand, by learning that more intervals of tasks have been completed to its left.

The variable *next* holds the next task process p is performing, or a point in an interval the process p is attempting to learn whether it has been completed.

The variable tmp holds values of pointers fetched from the shared memory through the *getHead*() or *getTail*() functions.

The variable *win* normally holds the output of the latest test-and-set function called on a location of vector W and is used for exiting the inner *while* loop.

Description of algorithm ARTA **for process** p**.** Initially process p sets the FREE set to contain all tasks and sets the *size* variable to n which is the total number of tasks. As long as there are more tasks to perform ($size > 0$) process p executes first function $progress(\text{FREE}, \text{DONE}, size)$ and then function $defrag(\text{FREE}, \text{DONE}, size)$:

In function $progress()$, process p picks a task to perform uniformly at random from the FREE set, using the $randomElement()$ function. Process p first performs the task through the $\mathsf{do}_{p,i}$ action and then calls the test-and-set operation associated with task i. If process p wins the test-and-set operation, it starts working on an interval of tasks. The interval starts at the randomly selected task i and moves to the right. Process p performs one task at a time, moving rightwards and then checks the associated test-and-set operation. As long as process p wins the test-and-set operations associated with the tasks, p keeps expanding the interval from the right side.The interval ends when the first test-and-set operation fails. When this happens, process p removes the interval from the FREE set, adds the interval to the DONE set and returns the sets FREE, DONE and the size of the FREE set. Note also that process p keeps the shared memory updated with the interval it is currently working at (or has just finished working), through the *head* and *tail* pointers of the W vector.

If the test-and-set operation on the randomly selected task fails, process p reads the *tail* pointer of the respective shared memory location and if it contains a valid value, it reads the *head* pointer from the $W[tail]$ location of the shared memory. Then it removes from the FREE set, the interval indicated by the *head* and *tail* pointers, adds the interval to the DONE set and returns the sets FREE, DONE and the size of the FREE set. Essentially process p detects that it collided in the specific task with another process and attempts to

increase the knowledge it has about jobs that have already been performed, by learning the interval of tasks that have been completed around the position of collision. Function *progress*() can result in the addition of at most one new leaf in the tree structures for the FREE and DONE sets.

In function *defrag*(), process p picks an interval from the set DONE. This is done through the use of the function *randomInterval*(). The interval is denoted by the $left$ and *right* pointers that indicate the boundaries of the interval. Process p repeatedly checks if there are completed tasks to the left of the interval, and if there are, it gathers the information pertaining to the interval that contains these tasks, and expands the interval to the left. This may result in merging intervals in the DONE set if all the tasks between the intervals have been performed. Specifically process p checks if the task $left-1$ has been performed, by reading the value in the $W[left-1]$ element of vector W. If the task has been performed, process p reads the $tail$ pointer of the respective shared memory location and then the *head* pointer from the $W[tail]$ location of the shared memory. Then it removes the interval indicated by the $tail$ and *head* pointers from the FREE set and adds it to the DONE set. The later returns a new set of $left$ and *right* pointers that indicate the boundaries of the interval, that includes the interval that was just added to the DONE set. This process is repeated, till the task $left-1$ has not been performed, or it is not marked as performed, at which point function *defrag*() returns the sets FREE, DONE and the size of the FREE set. Because tasks are performed in intervals that start at some point and are expanded in a rightwards direction, if the task at the left of any interval in the DONE set is performed, this task belongs in a new interval of completed tasks, that cannot expand any more. Also, the tasks corresponding to the left element of any interval in the DONE set, indicate the beginning of an interval of performed tasks.

8.2 Preliminaries

In this section we present the probabilistic arguments that we will later use in order to analyze the work complexity of algorithm ARTA. Let us assume that we have n ordered elements, partitioned in at most k groups. Groups are named g_i for $i \in \{1,\ldots,k\}$ with the i-th group containing d_i elements; then $\sum_{i=1}^{k} d_i = n$. Each element belongs to exactly one group. Elements are ordered and this order is maintained within the groups. We index the elements in group g_j with group indexes ind_{g_j} where $ind_{g_j} \in \{1,\ldots,d_j\}$, according to their order. Moreover, the global ordering is maintained across groups. Specifically for all i, j with $i < j$, any element in group g_i precedes any element in group g_j. We sample $m = \mathrm{O}(k \log k)$ elements uniformly at random with replacement (each element is sampled with probability $1/n$ in each coin flip).

We define as a *repartitioning*, an assignment of the n elements to a new set of groups performed in the following way: given the initial groups g_i and the $m = \mathrm{O}(k \log k)$ elements sampled uniformly at random with replacement, we select up to k unique elements from the set of m sampled elements together with the up to k starting elements of the original groups (that is all the elements from the groups which have group index 1).
Based on this selection of elements, we form the set of new groups $g'_1, g'_2, \ldots$ as follows. Consider g_i an existing group. If no element is selected from g_i, then all elements in the group g_i are *eliminated*. If we select $\mu \geq 1$ elements with group indexes $j_1 < j_2 < \ldots < j_\mu$ within g_i, we define μ new groups formed from the initial group g_i. The group starting at j_1 contains the elements with $j_1 \leq ind_{g_i} < j_2$, the group starting at j_2 contains the elements with $j_2 \leq ind_{g_i} < j_3$ etc. The last group starts at j_μ and contains the elements with $j_\mu \leq ind_{g_i} \leq d_i$ (remember that group g_i has d_i elements). If $j_1 > 1$, then the elements with $1 \leq ind_{g_i} < j_1$ are also eliminated.
We index the elements in the new groups g'_j in a similar way as before using group indexes

$ind_{g'_j} \in \{1,\ldots,d'_j\}$, where $d'_j = |g'_j|$. Note that each element in a new group g'_i can be mapped to exactly one element in the old groups.

Finally, if an element with $ind_{g'_j} = i$ from a new group g'_j was among the m coin flips but it was not among those selected for the formation of the new groups, we say that elements in g'_j with index $ind_{g'_j} \geq i$ are *eliminated* by the specific coin flip. In a similar way, we say that the elements in g'_j with index $ind_{g'_j} < i$, as well as all the elements in groups g'_w with $w \neq j$ *survive* that particular coin flip. An element *survives* the *repartitioning* if it was not been eliminated in any of the ways mentioned above, i.e., either (i) eliminated by some coin flip, or (ii) because the group it belonged was entirely eliminated in the repartitioning, or finally (iii) it belonged in the initial segment of a group that was repartitioned.

We will next show that if n ordered elements are partitioned into k groups maintaining order, for any small constant c and constant a, there exists constant C, such that with probability $1-k^{-a}$ for $Ck\log k$ coin flips there exists no *repartitioning* into up to k groups, such that more than $c \cdot n$ elements survive.

We start with a lemma that will prepare us for the main theorem of this section.

Lemma 8.1. *For any constants c, C, consider n ordered elements from which we sample $C \cdot k \cdot \log k$ elements uniformly at random with replacement. For a fixed element with index i among the n elements, the probability that we sample no element from the interval of elements with indexes $\left\{i, i+1, \ldots, i+\frac{c \cdot n}{k} - 1\right\}$ is less than or equal to $k^{-c \cdot C}$.*

Proof. The probability that an element from the interval $S_i = \left\{i, i+1, \ldots, i+\frac{c \cdot n}{k} - 1\right\}$ is sampled in a particular coin flip is:

$$\Pr[\text{element from } S_i \text{ is selected}] = \frac{\frac{c \cdot n}{k}}{n} = \frac{c}{k}$$

So the probability that no element from S_i is selected after $C \cdot k \cdot \log k$ coin flips is:

$$\Pr[\text{no element from } S_i \text{ is selected after } C \cdot k \cdot \log k \text{ coin flips}] = \left(1 - \frac{c}{k}\right)^{C \cdot k \cdot \log k} \leq e^{-c \cdot C \cdot \log k} \leq k^{-c \cdot C}.$$

We remark that the above result of the theorem is essentially tight: If we sample only $C \cdot k$ elements for some C then it holds that the probability of choosing no element in S_i will be $(n - cn/k)^{C \cdot k}/n^{C \cdot k} = (1 - c/k)^{C \cdot k} = ((1 - c/k)^k)^C$ which approximates the constant $e^{-c \cdot C} > 0$ for large values of k. □

We are now ready to state the main theorem of this section.

Theorem 8.2. *If n ordered elements are partitioned into k or less groups maintaining order, for any small constant c and constant a, there exists constant C, such that with probability $1 - k^{-a}$ after $C \cdot k \cdot \log k$ coin flips there exists no repartitioning into up to k groups, such than more that $c \cdot n$ elements survive.*

Proof. Let S be the set of all the starting points of the initial groups and all the distinct elements in the $C \cdot k \log k$ coin flips. Each element in S can be uniquely defined by the tuple $[j,i]$, where g_j the group at which the element belongs and $i = ind_{g_j}$ the group index of the element. Clearly $|S| \leq k + C \cdot k \log k$. The probability that for an element $[j,i] \in S$, $d_j \geq i + \frac{c \cdot n}{k} - 1$ and no element is sampled from the interval of elements in group g_j that starts at i and ends at $i + \frac{c \cdot n}{k} - 1$ is by Lemma 8.1 less that or equal to $k^{-c \cdot C}$.

We define A to be the event that there exists element $[j,i] \in S$, such that $d_j \geq i + \frac{c \cdot n}{k} - 1$ and no element is sampled from the interval of elements in group g_j that starts at i and ends at $i + \frac{c \cdot n}{k} - 1$. By applying the union bound we have the following:

$$\Pr[\mathrm{A}] \leq (C \cdot k \log k + k) \cdot k^{-c \cdot C} \leq k^{-(c \cdot C - 2)}$$

for sufficient large k. Observe now that for any choice of $C \geq \frac{a+2}{c}$ we have that:

$$\Pr[\mathrm{A}] \leq k^{-a}$$

We want to find a repartitioning such that more than $c \cdot n$ elements survive. In order to do the repartitioning we will have to select up to k elements from the set S. From the above inequality, with probability greater than $1 - k^{-a}$ there exists no element in $[j,i] \in S$ such that if selected as a starting point for a group g'_w, in the repartitioning more than $\frac{c \cdot n}{k}$ elements will survive in the group g'_w. Thus, with probability greater than $1 - k^{-a}$ there exists no repartitioning such that more than $c \cdot n$ elements survive. □

8.3 Analysis of Algorithm ARTA

We will start with the correctness analysis of algorithm ARTA and then proceed to the high probability work complexity. Remember that for correctness in the Write-All problem we need to prove that all non-failed processes terminate, that when the first non-failed process terminates all tasks have been performed and that any process that terminates knows that all tasks have been performed according to Definition 2.9.

8.3.1 Correctness

For the correctness analysis we will start with a lemma that states that if a task has been added to the DONE set of a process p, this task has been performed by some process. Then it will be easy to show that algorithm ARTA correctly solves the Write-All problem.

Lemma 8.3. *In any execution α of algorithm* ARTA*, if at state s for some process p and some task i, $i \in$* DONE *for the* DONE *set of process p, then there exists action $\mathsf{do}_{q,i}$ that precedes state s in α, for some process q (p and q can be the same process).*

Proof. For any process p the DONE set is initially empty. Elements are added to the DONE set either during the execution of line 26 of function *progress*() or during the execution of line 15 of function *defrag*().

We first examine the case where i has been added to the DONE set during the execution of line 15 of function *defrag*(). This means that there exists in α call to DONE.*add*(*tail*, *head*) by process p that precedes state s, such that $tail \leq i \leq head$. We have two cases depending on whether $tail = head$ or not.

Case 1: $tail = head$. This means that $i = tail = head$. Moreover the check on the while statement in line 2 for the while loop that resulted in the call of DONE.*add*(*tail*, *head*), was done on $W[i]$ and returned true. From the above we have that some process q must have performed a successful $W[i].tas()$ either in line 3 or line 13 of function *progress*(). This is preceded by $\mathsf{do}_{q,i}$ either in line 2 or line 12 respectively. This $\mathsf{do}_{q,i}$ precedes execution of line 15 of function *defrag*() and thus precedes state s.

Case 2: $tail \neq head$. This means that the if statement in line 11 returned true.

In order to see why this happens, let's observe that if both *if* statements returned false, from lines 4 and 5 of function *defrag*(), we have that $tail = head$. Moreover, if the *if* statement in line 11 returned false, it must be the case that the if statement in line 7 returned false as well. This happens because, the head and tail pointers in a shared memory location, are only set in lines 8 and 9 in function *progress*(). Since we first set the head pointer in the tail of the interval and then the tail pointer in the head of the interval, if the head pointer in the tail of an interval is still $\perp$, it must be the case that the tail pointer in the head of the interval is also $\perp$.

From the discussion above we have that the *if* statement in line 11 returns true. Since only in the tail of an interval the head pointer has a value different than $\perp$, it must be the case that the $W[tail]$ in line 10, is the tail of a working interval. The value returned by $W[tail].getHead()$ in line 10, can only be set in line 8 of function *progress*() by some process q. Process q must have won all the test end set operations in the interval defined by *tail* and *head*, either in line 3 for *tail* or in line 13 of the test and set in $(tail, head]$. An

invocation of a test and set operation in position $j \in [tail, head]$, is preceded by a $do_{q,j}$ operation. These *do* operations precede the execution of line 8 of function *progress*() by process q, that set the $W[tail]$ to the value returned by the execution of $W[tail].getHead()$ in line 10 of function *defrag*(). So there exists action $\mathsf{do}_{q,i}$ that precedes state s.

Next we examine the case where i has been added to the DONE set during the execution of line 26 of function *progress*(). If the *if* statement in line 6 returned true, it is easy to see that there exists action $\mathsf{do}_{p,i}$, that precedes the execution of line 26 of function *progress*(), so action $\mathsf{do}_{p,i}$ precedes state s.

If the *if* statement in line 6 returned false, then with similar arguments as for the case of line 15 of function *defrag*(), we have that for some process q, there exists action $\mathsf{do}_{q,i}$ that precedes state s. □

We can now proceed to the main theorem of this subsection.

Theorem 8.4. *Algorithm* ARTA *solves the Write-All problem.*

Proof. We will start by showing that any participating process p that does not crash in an execution α, eventually terminates.

It is easy to see that in every call of function *progress*() by process p at least one element is removed from the FREE set and at least one element is added in the DONE set. Moreover any element that is removed from the FREE set, is added to the DONE set. The FREE set has initially n elements. Since there are finite elements in the FREE set and at least one element is removed in each call of function *progress*(), if process p participates in algorithm ARTA and does not crash, there exists state s in α, such that for all states in $s' \in \alpha$ that are preceded by s, the FREE set for process p is empty. The first time after state s that line 4 will be executed in the driver script for process p, process p will get out of the while loop and terminate. When this happens, the FREE set for process p will be

empty and the DONE set will contain all the tasks, specifically DONE $= [0, n-1]$. From Lemma 8.3, we have that since for some process p, DONE $= [0, n-1]$, all tasks have been preformed by some process.

Moreover, since a process q terminates only if it gets out of the loop in lines $4 - 7$ of the driver script, we have that the FREE set for process q is empty and as a result the DONE set is equal to $[0, n-1]$. So all tasks have been performed by some process and process q knows that all tasks have been performed. Thus algorithm ARTA correctly solves the Write-All problem. □

8.3.2 High Probability Work Complexity

Next we proceed with the work complexity analysis of algorithm ARTA. For the purposes of the analysis we split the execution of the algorithm in phases and seasons. We have two seasons, each split into multiple phases. The definition of seasons is based on how many tasks are available at the beginning of the phases of the season. In the first season we have more than $O(k^2 \log^2 k)$ available tasks, in the second season we have less than or equal to $O(k^2 \log^2 k)$ available tasks. A phase is an execution fragment α_i' (execution for the very first phase) that starts at state s_i and ends at state s_i'. We say that in phase α_i' with starting state s_i there are $|s_i.\text{FREE}|$ available tasks, where $s_i.\text{FREE}$ is the set of tasks at state s_i for which no test-and-set operation has terminated returning TRUE in the execution that ends with state s_i. Given a state s, we define a *current* coin flip for state s, as the invocation of a test-and-set operation in line 3 of function *progress*() that is preceded by state s, if the test-and-set operation corresponds to a task that belongs in the set of available tasks s.FREE. Similarly, in phase α_i' we define a *current* coin flip, as a *current* coin flip for state s_i in the execution fragment α_i'. Note that the execution of line 1 of function *progress*() (the actual coin flip) could have taken place before state st_i, but we

are using as a linearization point the test-and-set operation, since it is an atomic operation that affects the shared memory. We say that a current coin flip of a phase α_i' is *completed* in the same phase if the execution of line 3 of function *progress*() returned FALSE or if it returned TRUE but a latter execution of line 13 of function *progress*() returned FALSE as a result of the test-and-set operation, in some action of the execution fragment α_i'.

A phase both in season 1 and in season 2 is comprised by $C \cdot k \log k$ consecutive *current* coin flips, for a properly selected constant C (the constant depends on the definition of high probability $1 - k^{-a}$ and is selected as a function of constant a). Specifically, line 3 in the algorithm involves the execution of an atomic operation that interacts with the shared memory. Such operations are linearizable. The end of a phase s_i' is the resulting state after the last ($C \cdot k \log k$-th) current coin flip. The same state defines the beginning of the next phase. The first phase in the algorithm starts with the starting state of the algorithm, while the last phase ends with the ending state of the algorithm (the last phase may have less than $C \cdot k \log k$ *current* coin flips). Clearly between two consecutive phases there can exist no more than k *current* but not completed coin flips (since there are k participating processes in the algorithm).

Season 1 starts at the beginning of the algorithm if $n > k^2 \log^2 k$, otherwise the algorithm starts directly at season 2. Season 2 starts at the first phase for which there are less or equal to $k^2 \log^2 k$ available tasks.

For the analysis we are going to use the following approach. First we will show that in season 1 between consecutive phases the available tasks are reduced by at least $O(k^2 \log^2 k)$ tasks with high probability. Then we will show that in season 2 the available tasks are halved between consecutive phases. This will give us an upper bound on the number of current coin flips. We will argue that tasks are fragmented into intervals by the current coin flips and thus we will get an upper bound on the number of intervals that can

be created in an execution. Next we will show that in a specific phase, coin flips that are not current, only learn about intervals that have been created in previous phases. This will allow us to bound the number of such coin flips (based on the number of intervals). We will then show that intervals that are discovered during the invocation of function *defrag*() are only discovered once, which will also give us an upper bound on the work needed in order to learn about completed tasks, during invocations of function *defrag*(). Finally we will show that the sets FREE and DONE that a process p keeps in its internal memory are with high probability fragmented in no more than $O(k)$ intervals, which will give us an upper bound on the work needed to access those sets, which will allow us to complete the work complexity analysis.

We will start with a lemma that bounds the number of intervals, tasks are separated into, in the shared memory. This will help us with the rest of the analysis.

Lemma 8.5. *For all executions of algorithm* ARTA *where k processes participate, the available tasks are split into no more than k intervals.*

Proof. The k processes that participate in algorithm ARTA initially create at most k working intervals of tasks and thus, there are at most k intervals of *available* tasks (note that we consider that tasks wrap around, that is the n-th task continues with the first task). Each process p stops working on its current interval at some task i if it loses the test-and-set operation on i (i now denotes the end of the interval).We will show that the number of working intervals does not increase or equivalently the number of intervals of *available* tasks does not increase.

If process p loses a test-and-set operation at task i, it must be the case that some other process q has won the test-and-set operation at i and is working (or was working) on an interval starting at position i. The two intervals are merged and although process p might create a new working interval, the number of working intervals does not increase. Thus

we have that at most k intervals of *available* tasks can exist for any state s in any execution α of algorithm ARTA. □

Next we will show that during a phase in season 1 more than $O(k^2 \log^2 k)$ tasks are being performed with high probability. This will allow us to bound the number of intervals that can be formed during season 1.

Lemma 8.6. *For a state s, if* $|s.\text{FREE}| > k^2 \log^2 k$*, for any constant a and c there exists constant C, such that for any state* s' *after* $C \cdot k \log k$ current *coin flips,* $|s.\text{FREE}| - |s'.\text{FREE}| \geq ck^2 \log^2 k$ *with probability greater that* $1 - k^{-a}$.

Proof. Let $\bar{s}$ be the resulting state after the last ($C \cdot k \log k$-th) current coin flip. The sets of available tasks s.FREE and $\bar{s}$.FREE at states s and $\bar{s}$ (resp.) are each split into at most k intervals from Lemma 8.5. The at most k intervals of tasks can be seen as groups of ordered elements. Essentially set $\bar{s}$.FREE is a subset of an up to k group *repartitioning* of set s.FREE and $|\bar{s}.\text{FREE}|$ is less than the number of elements that survive the repartitioning. The ideal strategy for the adversary (in terms of minimizing the number of tasks performed by the $C \cdot k \log k$ current coin flips) would be to make $\bar{s}$.FREE, the repartitioning that maximizes the number of elements that survive. We will use Theorem 8.2 in order to bound the number of elements in $\bar{s}$.FREE.

In a current coin flip at task i by a process p, p either wins the test-and-set operation at line 3 of function *progress*() or not. In the first case, the interval at which the task i belonged is now split in two new intervals. One ending at task $i-1$ and one starting at task i. If this split is maintained at state s', it is equivalent with selecting i as the beginning of a group in the repartitioning.

In the second case, there exists process $q \neq p$ that has performed task i. This can happen in two ways. Either process q was positioned at the beginning of the interval

task i belonged and has performed all tasks in the interval up to task i, or process q has performed a coin flip inside that interval task i belongs, in a task with index in the interval less than or equal to the index of i (splitting the interval in two) and then performed all tasks between the task it started at and task i.

From the discussion above, set $\overline{s}$.FREE is a subset of a repartitioning of set s.FREE, since it consists of the intervals of available elements at state $\overline{s}$. Specifically, an interval of available tasks in $\overline{s}$ cannot contain any task for which there has been a current coin flip in α'. So each such interval is a subset of a group for some repartitioning of s.FREE given the $C \cdot k \log k$ current coin flips. There are up to k such intervals, thus $\overline{s}$.FREE is a subset of some repartitioning of s.FREE, for the $C \cdot k \log k$ current coin flips.

We want to prove that , $|s.\text{FREE}| - |\overline{s}.\text{FREE}| \geq ck^2 \log^2 k$ with probability greater that $1 - k^{-a}$.

So we want to show that:

$$|s.\text{FREE}| - |\overline{s}.\text{FREE}| \geq ck^2 \log^2 k$$

$$|\overline{s}.\text{FREE}| \leq |s.\text{FREE}| - k^2 \log^2 k \leq (1-c)k^2 \log^2 k$$

As we discussed above $\overline{s}$.FREE is a subset of some repartitioning of s.FREE, for the $C \cdot k \log k$ current coin flips. Thus from Theorem 8.2 we have that for any constants a and c', if we choose $C > \frac{a+1}{c'}$, we have that:

$$|\overline{s}.\text{FREE}| \leq c'|s.\text{FREE}| \leq c'k^2 \log^2 k$$

with probability greater than or equal to $1 - k^{-a}$.

Thus for $C > \frac{a+1}{1-c}$ we have that:

$$|\overline{s}.\text{FREE}| \leq (1-c)k^2 \log^2 k$$

with probability greater than or equal to $1 - k^{-a}$.

□

From the previous lemma, using $C > \frac{a+1}{1-c}$ in the definition of a phase we have for season 1 the following lemma.

Lemma 8.7. *For any phase α_i in season 1 we have that $|s_i.\text{FREE}| - |s_i'.\text{FREE}| \geq ck^2 \log^2 k$ with probability greater than $1 - k^{-a}$.*

Proof. For any phase α_i in season 1 we have that $|s_i.\text{FREE}| > k^2 \log^2 k$. In phase α_i we have $C \cdot k \log k$ current coin flips, where $C > \frac{a+1}{1-c}$ so for the last state s_i' of phase α_i from Lemma 8.6 we have that $|s_i.\text{FREE}| - |s_i'.\text{FREE}| \geq ck^2 \log^2 k$ with probability greater that $1 - k^{-a}$.

□

Next we will show that during a phase in season 2 the available tasks are being halved with high probability. This will allow us to bound the number of intervals that can be formed during season 2.

Lemma 8.8. *For a state s, if $|s.\text{FREE}| \leq k^2 \log^2 k$, for any constant a and c there exists constant C, such that for any state s' after $C \cdot k \log k$* current *coin flips, $|s'.\text{FREE}| \leq c \cdot |s.\text{FREE}|$ with probability greater that $1 - k^{-a}$.*

Proof. Let $\overline{s}$ be the resulting state after the last ($C \cdot k \log k$-th) current coin flip. The sets of available tasks s.FREE and $\overline{s}$.FREE at states s and $\overline{s}$ (resp.) are each split into at most k intervals from Lemma 8.5. The at most k intervals of tasks can be seen as groups of

ordered elements. With similar arguments as in Lemma 8.6 we have that $\bar{s}$.FREE is a subset of some repartitioning of s.FREE, for the $C \cdot k \log k$ current coin flips. Thus from Theorem 8.2 we have that for any constants a and c, if we choose $C > \frac{a+1}{c}$, it holds that:

$$|\bar{s}.\text{FREE}| \leq c \cdot |s.\text{FREE}|$$

with probability greater than or equal to $1 - k^{-a}$.

For any state $s' > \bar{s}$, it holds that $|s'.\text{FREE}| \leq |\bar{s}.\text{FREE}| \leq c \cdot |s.\text{FREE}|$, since the set available task never increases in the execution of algorithm ARTA.

□

From the previous lemma, using $C > \frac{a+1}{c}$ in the definition of a phase we have for season 2 the following lemma.

Lemma 8.9. *For any phase α_i in season 2 we have that $|s'_i.\text{FREE}| \geq c \cdot |s_i.\text{FREE}|$ with probability greater than $1 - k^{-a}$.*

Proof. For any phase α_i in season 2 we have that $|s_i.\text{FREE}| \leq k^2 \log^2 k$. In phase α_i we have $C \cdot k \log k$ current coin flips, where $C > \frac{a+1}{c}$ so for the last state s'_i of phase α_i from Lemma 8.8 we have that $|s'_i.\text{FREE}| \leq c|s_i.\text{FREE}|$ with probability greater that $1 - k^{-a}$.

□

If we choose $C > 2(a+1)$, Lemmas 8.7 and 8.9 hold for $c = \frac{1}{2}$. Now we are going to bound the total number of intervals that can be created in the execution of algorithm ARTA. This will also allow us to bound the total number of coin flips that can take place in any execution of ARTA by all participating processes k.

Lemma 8.10. *There exists constant δ such that season 1 has no more than $(1+\delta)\frac{2n}{k^2 \log^2 k}$ phases with probability greater than $1 - k^{-a}$.*

Proof. For any phase α_i' in season 1 we have that $|s_i.\text{FREE}| > k^2 \log^2 k$. Moreover from Lemma 8.7 for $C > 2(a+1)$ and $c = \frac{1}{2}$ we have that $|s_i.\text{FREE}| - |s_i'.\text{FREE}| \geq \frac{1}{2}k^2 \log^2 k$ with probability greater than $1 - k^{-a}$. Also from the definition of a phase, it holds that $s_i' = s_{i+1}$ for consecutive phases α_i' and α_{i+1}'. Finally at the first phase α_0' of season 1 it holds that $|s_0.\text{FREE}| = n$, since s_0 is the starting state of algorithms ARTA.

We say that a phase EX_i' in season 1 is successful if $|s_i.\text{FREE}| - |s_i'.\text{FREE}| \geq \frac{1}{2}k^2 \log^2 k$. Since in each successful phase in season 1 the available tasks are reduced by $\frac{1}{2}k^2 \log^2 k$, there can be no more than $\frac{2n}{k^2 \log^2 k}$ successful phases in season 1, since if we had $\frac{2n}{k^2 \log^2 k}$ successful phases in season 1 there will be no more available tasks left, which implies that we should have entered season 2.

If season 1 has more than $(1+\delta)\frac{2n}{k^2 \log^2 k}$ phases, it must be the case that at the first $(1+\delta)\frac{2n}{k^2 \log^2 k}$ phases, strictly less than $\frac{2n}{k^2 \log^2 k}$ were successful. As we discussed a phase in season 1 is successful with probability at least $1 - k^{-a}$ and is not successful with probability less than k^{-a}. Let A be the event that more than $\delta \frac{2n}{k^2 \log^2 k}$ phases were not successful in the first $(1+\delta)\frac{2n}{k^2 \log^2 k}$ phases of season 1. Let $N = \frac{2n}{k^2 \log^2 k}$. We have for the probability of event A:

$$\Pr[A] = \sum_{i=\delta N+1}^{(1+\delta)N} \binom{(1+\delta)N}{i} k^{-a \cdot i} \left(1 - k^{-a}\right)^{(1+\delta)N - i}$$

$$\Pr[A] \leq \sum_{i=\delta N+1}^{(1+\delta)N} \left(\frac{(1+\delta)Ne}{i}\right)^{i} k^{-a \cdot i} \left(1 - k^{-a}\right)^{(1+\delta)N - i}$$

Since $i > \delta N$ we have:

$$\Pr[A] \leq \sum_{i=\delta N+1}^{(1+\delta)N} \left(\frac{(1+\delta)e}{\delta}\right)^{i} k^{-a \cdot i} \left(1 - k^{-a}\right)^{(1+\delta)N - i}$$

$$\Pr[A] \leq \sum_{i=\delta N+1}^{(1+\delta)N} \left(\frac{(1+\delta)\,e}{\delta k}\right)^{i} k^{-a} \left(1-k^{-a}\right)^{(1+\delta)N-i}$$

Since δ constant and k the number of participating processes, there exists δ such that $\frac{(1+\delta)e}{\delta k} < 1$. We have that if $k \geq 9$ and $\delta > \frac{1}{2}$, $\frac{(1+\delta)e}{\delta k} < 1$. Moreover, since $\delta N < i \leq (1+\delta)N$, we have that $(1-k^{-a})^{(1+\delta)N-i} < 1$. So we have:

$$\Pr[A] \leq \sum_{i=\delta N+1}^{(1+\delta)N} \left(\frac{(1+\delta)\,e}{\delta k}\right)^{i} k^{-a} \leq \sum_{i=\delta N+1}^{(1+\delta)N} \left(\frac{(1+\delta)\,e}{\delta k}\right)^{\delta N} k^{-a} \leq N \left(\frac{(1+\delta)\,e}{\delta k}\right)^{\delta N} k^{-a}$$

There exists constant β such that for $n > \beta k^2 \log^2 k$, $\delta > 1$ and large enough k (the number of participating processes), we have that $N\left(\frac{(1+\delta)e}{\delta k}\right)^{\delta N} < 1$. So we have that:

$$\Pr[A] \leq k^{-a}$$

From the above, we get that season 1 has no more than $\frac{4n}{k^2 \log^2 k}$ phases with probability greater than $1-k^{-a}$.

□

Lemma 8.11. *There exists constant δ such that season 2 has no more than $3\,(1+\delta)\log k$ phases with probability greater than $1-k^{-a}$.*

Proof. For any phase α'_i in season 2 we have that $|s_i.\text{FREE}| < k^2 \log^2 k$. Moreover from Lemma 8.9 for $C > 2(a+1)$ and $c = \frac{1}{2}$ we have that $|s'_i.\text{FREE}| \leq \frac{1}{2}|s_i.\text{FREE}|$ with probability greater than $1-k^{-a}$. Also from the definition of a phase, it holds that $s'_i = s_{i+1}$ for consecutive phases α'_i and α'_{i+1}. Finally at the first phase α'_j of season 2 it holds that $|s_j.\text{FREE}| < k^2 \log^2 k$.

With similar arguments as in the proof of Lemma 8.11, we say that a phase α'_i in season 2 is successful if $|s'_i.\text{FREE}| \leq \frac{1}{2}|s_i.\text{FREE}|$. Since in each successful phase in season 2 the available tasks are halved, there can be no more than $3\log k$ successful phases in

season 2, since after $\log\left(k^2\log^2 k\right) < 3\log k$ successful phases there are no more available tasks left.

If season 2 has more than $3(1+\delta)\log k$ phases, it must be the case that at the first $3(1+\delta)\log k$ phases, strictly less than $3\log k$ were successful. As we discussed a phase in season 2 is successful with probability at least $1-k^{-a}$ and is not successful with probability less than k^{-a}. Let A be the event that more than $3\delta\log k$ phases were not successful in the first $3(1+\delta)\log k$ phases of season 2. We have for the probability of event A:

$$\Pr[A] = \sum_{i=3\delta\log k+1}^{3(1+\delta)\log k} \binom{3(1+\delta)\log k}{i} k^{-a\cdot i}\left(1-k^{-a}\right)^{3(1+\delta)\log k-i}$$

$$\Pr[A] \leq \sum_{i=3\delta\log k+1}^{3(1+\delta)\log k} \left(\frac{3(1+\delta)\log ke}{i}\right)^{i} k^{-a\cdot i}\left(1-k^{-a}\right)^{3(1+\delta)\log k-i}$$

Since $i > 3\delta\log k$ we have:

$$\Pr[A] \leq \sum_{i=3\delta\log k+1}^{3(1+\delta)\log k} \left(\frac{(1+\delta)e}{\delta}\right)^{i} k^{-a\cdot i}\left(1-k^{-a}\right)^{3(1+\delta)\log k-i}$$

$$\Pr[A] \leq \sum_{i=3\delta\log k+1}^{3(1+\delta)\log k} \left(\frac{(1+\delta)e}{\delta k}\right)^{i} k^{-a}\left(1-k^{-a}\right)^{3(1+\delta)\log k-i}$$

Since δ constant and k the number of participating processes, there exists δ such that $\frac{(1+\delta)e}{\delta k} < 1$. So if $k \geq 9$ and $\delta > \frac{1}{2}$, $\frac{(1+\delta)e}{\delta k} < 1$. Moreover, since $3\delta\log k < i \leq 3(1+\delta)\log k$, we have that $\left(1-k^{-a}\right)^{3(1+\delta)\log k-i} < 1$. So we have:

$$\Pr[A] \leq \sum_{i=3\delta\log k+1}^{3(1+\delta)\log k} \left(\frac{(1+\delta)e}{\delta k}\right)^{i} k^{-a} \leq \sum_{i=3\delta\log k+1}^{3(1+\delta)\log k} \left(\frac{(1+\delta)e}{\delta k}\right)^{3\delta\log k} k^{-a}$$

$$\Pr[A] \leq 3\log k\left(\frac{(1+\delta)e}{\delta k}\right)^{3\delta\log k} k^{-a}$$

For $\delta > 0.6$ and $k > 9$ we have that $3\log k \left(\frac{(1+\delta)e}{\delta k}\right)^{3\delta \log k} < 1$. Thus we have:

$$\Pr[A] \leq k^{-a}$$

From the above, we get that season 2 has no more than $5\log k$ phases with probability greater than $1 - k^{-a}$. □

Now we can bound the number of coin flips (calls to the function *progress*) that can be performed by the k participating processes in algorithm ARTA.

Lemma 8.12. *In any execution of algorithm* ARTA *there exist no more than* $\frac{4(C+2)n}{\log k} + 5(C+2)k^2\log^2 k$ *calls to the function progress*() *with very high probability.*

Proof. Each call of function *progress*() results in exactly one coin flip. The way we have defined a phase in an execution α, a coin flip can either be current or not. Each phase has exactly $Ck\log k$ current coin flips, apart from the last phase of season 2 which may have less current coin flips, so that each current coin flip, happens in exactly one phase of the execution. In season 1 we have less than $\frac{4n}{k^2\log^2 k}$ phases with probability $1 - k^{-a}$, from Lemma 8.10 . Moreover in season 2 we have less than $5\log k$ phases again with probability $1 - k^{-a}$, from Lemma 8.11. From the definition of season 2 and from Theorem 8.4, in the last state of the last phase of season 2 all non-failed processes have terminated. From the above we have that any execution α of ARTA has less than $\frac{4n}{k^2\log^2 k} + 5\log k$ phases with probability greater than $(1 - k^{-a})^2 > 1 - 2k^{-a}$.

In each phase α_i' we have $Ck\log k$ current coin flips (or less). Each current coin flip may result in the creation of a new active working interval. Moreover, there are up to k active working intervals at the initial state s_i that have been created at previous phases. For the purposes of the analysis, we consider that these intervals have been completed, and that the process that was working on them, now works on a new interval, that starts

at the first available task (in state s_i) at the right of the interval. Although the intervals are one in practice (that is a process that learns about a completed task of the new or the old working interval, as a result of a coin flip in a state $t > s_i$, will learn about the tasks that have been completed up to state t in both intervals), we now treat the intervals as two different intervals. One that is completed (it expands no more to its right) and one that is an active working interval, that expands to its right side. That means that a process, in order to learn the completion of all tasks of the interval needs now to learn about one task of the old interval and one of the new interval (as a result of two different coin flips). For the old interval it will learn about all the tasks of the interval which is not expanding any more. On the other hand for the new interval it will learn about the tasks that have been performed so far in the new interval, but not about the tasks performed in the old interval. As a result of this convention, during a phase α_i', current coin flips could return available tasks, completed tasks that belong in active intervals or completed tasks that belong in completed intervals that have been created and completed in phase α_i'. On the other hand, coin flips that are not current can return only tasks that belong in completed intervals that have been created during a previous phase α_j, with $j < i$.

From the above we have that at the end of the execution, the set of completed tasks is fragmented in less than $\left(\frac{4n}{k^2 \log^2 k} + 5\log k\right)(Ck\log k + k) = \frac{4(C+1)n}{k\log k} + 5(C+1)k\log^2 k$ intervals, with probability greater than $1 - 2k^{-a}$. As we discussed above, during a phase α_i' a coin flip, performed by a process p, that is not current returns a task that belongs in a completed interval that has been created during a previous phase α_j, with $j < i$, by some process $q \neq p$. As a result of the non current coin flip, process p learns about all the tasks that have been performed by process q at the completed interval that includes the returned task, and will not perform any coin flips that will return tasks in the specific completed interval in subsequent states. So a participating process p can perform no more

than $\frac{4(C+1)n}{k\log k} + 5(C+1)k\log^2 k$ non current coin flips. Since we have a total number of k participating processes, in any execution α of ARTA, there are no more than $\frac{4(C+1)n}{\log k} + 5(C+1)k^2\log^2 k$ non current coin flips. If we add the total number of current coin flips, in the non current coin flips, we get the total number of calls to the function *progress*. So we have that in any execution α of ARTA, there are no more than $\frac{4(C+2)n}{\log k} + 5(C+2)k^2\log^2 k$ calls to the function *progress*, with probability greater than $1-2k^{-a}$. □

So far we have bounded the total number of calls to the function *progress*() (and as a result to function *defrag*()). In order to complete the work complexity analysis, we need to bound the work needed for each call. This cost is dominated by the work needed for the calls of the operations on the FREE and DONE sets (that are stored internally by each process). Each call needs work $O(\log|\text{FREE}|) = O(\log|\text{DONE}|)$, assuming that the sets are stored in some efficient tree data structure. We will show that the size of these two sets is $O(k)$ with high probability. The idea is that in ARTA, every call to function *progress*() is followed by a call to function *defrag*(). The call to function *progress*() adds exactly 1 element to each of the sets (a new interval of completed tasks). The call to function *defrag*(), probabilistically removes elements from the two sets. Each call to *defrag*(), removes on expectation more than $\frac{|\text{FREE}|}{2k}$ elements. That means that as the size of the set increases, more elements are removed on expectation in each call. For the purposes of our analysis, we will only use the fact that the probability of removing at least two elements in a call of function *defrag*() is considerably higher of the probability of removing less than two elements, if the size of set FREE is large.

In order to understand what happens during an invocation of function *defrag*(), we first examine the DONE set. The DONE set is composed of non-intersecting intervals of completed tasks. So each element is defined by *tail* and *head*, the first and last task of the interval. The intervals in DONE contain consecutive tasks that process p knows that some

process has completed (remember that set DONE is in the local memory of process p). From Lemma 8.5 we have that the available tasks (and as a result the completed tasks) are split into no more than k intervals. So each interval of completed tasks in set DONE belongs in exactly one bigger interval of completed tasks and there are no more than k such bigger intervals. Moreover the bigger intervals are again comprised of consecutive tasks and they are not intersecting. So we could see the intervals in set DONE as ordered elements that are logically partitioned in upto k groups. This ordering is maintained within the groups, and the global ordering is maintained across groups, in a way similar with what we had in Section 8.2. This means that if we pick any two groups, all the elements of one group either precede or are preceded by all the elements of the other group. Now during an invocation of function *defrag*(), an element of the set DONE is picked uniformly at random at line 1 by the DONE.*randomInterval*() function. As mentioned above this element represents an interval of completed tasks that belongs in a bigger interval of completed tasks. At the *while* loop (lines 2 - 16) all the tasks that belong in that bigger interval and are to the left of the interval picked by the DONE.*randomInterval*() function, are discovered and merged with that interval. As a result, all elements (intervals of tasks) of set DONE that belong in the same group (bigger interval) as the element (interval of tasks) we picked and precede that element (contain completed tasks to the left of the tail of the selected interval), are merged in to a new interval. As a result the cardinality of set DONE is decreased by the number of elements in the group of the randomly selected element, that precede this element. We say that the *defrag*() operation *removed* those elements. If from a group we select the leftmost element, then the cardinality of set DONE remains unchanged at the end of function *defrag*(). Since there are at most k such groups, there are less than or equal to k such leftmost elements. If we pick the second from the left element of a group, then the cardinality of set DONE is decreased by 1 at the end of

function *defrag*(). Again there are less than or equal to k such elements. In any other case, the cardinality of set DONE is decrease by 2 or more. Remember that each call of function *progress*() results in the addition of at most 1 new element in set DONE. We are ready now to bound the size of set DONE. Remember that the set is initially empty (has no elements).

Lemma 8.13. *In any execution* α *of algorithm* ARTA *for any process* p *and any state* $s \in \alpha$, $|s.\text{DONE}_p| \leq 20k$ *with high probability.*

Proof. Let's examine the set DONE for a specific process p. As discussed above every call to function *progress*() results in the addition of at most one new element in the DONE set. This call is then followed by a call to function *defrag*(). A call to function *defrag*() either reduces the number of elements in the DONE set, or leaves the same of elements in the DONE set. So in order to have a state $s \in \alpha$ such that $|s.\text{DONE}_p| > 20k$, there must exist a state $s' \in \alpha$ that precedes s in execution α such that $|s'.\text{DONE}_p| = 10k$ and for all states t in the execution fragment α' that begins with state s' and ends at state s, $|t.\text{DONE}_p| \geq 10k$. Moreover in this execution fragment if there are $i+1$ calls to function *progress*(), there must exist at least i calls to function *defrag*(). Let us assume that all the $i+1$ calls to function *progress*(), result in the addition of a new element to set DONE_p. It must be the case that $i \geq 10k$.

From the discussion above the probability that the $j-th$ call to function *defrag*() removes one or no elements is $\frac{2k}{|\text{DONE}_p|} \leq \frac{1}{5}$, since $|\text{DONE}_p| \geq 10k$. We name this event X_j. This event either removes one element from DONE_p or removes no elements. For simplicity we assume that as a result of event X_j no element is removed. In all other cases 2 or more elements are removed. Again for simplicity we assume that exactly 2 elements are removed in the event $\bar{X_j}$. We are interested on the event $X = \sum_{j=1}^{i} X_j$. Since the events X_j are independent and $\Pr[X_j] \leq 0.2$, for the expectation of X we have $\text{E}[X] \leq 0.2i$.

In order for $|s.\text{DONE}_p| > 20k$ it must be the case that $X \geq 0.5i + 10k$, since this means that less than $0.5i - 10k$ events $\bar{X}_j$ take place, which result in the removal of less that $i - 10k$ elements from set DONE_p. Since $i+1$ elements have been added from function *progress*(), we have that $|s.\text{DONE}_p| > 20k$. From Chernoff bounds we have:

$$\Pr[X \geq \text{E}[X] + \lambda] \leq e^{-\frac{2\lambda^2}{i}}$$

From which for $\text{E}[X] \leq 0.2i$ and $\lambda = 0.3i + 10k$ we have:

$$\Pr[X \geq 0.5i + 10k] \leq e^{-\frac{2(0.3i+10k)^2}{i}} \leq e^{-0.18i}$$

We call Y_i the event $X \geq 0.5i + 10k$, so from the discussion above we have that $\Pr[Y_i] \leq e^{-0.18i}$. In any execution α a process p can make no more than n calls to function *progress*() from Theorem 8.4. For simplicity let $|\text{DONE}| \geq 10k$ for all states of execution α. This increases the probability of existence of a state $s \in \alpha$ such that $|s.\text{DONE}_p| > 20k$. So during any execution α the probability that there exists state $s \in \alpha$ such that $|s.\text{DONE}_p| > 20k$ is given by event $Y = \bigcup_{i=10k}^{n} Y_i$, since we need at least $10k$ calls of function *process* in order to have the cardinality of set DONE increased by $10k$. For the probability of event Y we have:

$$\Pr[Y] = \Pr[\bigcup_{i=10k}^{n} Y_i] \leq \sum_{i=10k}^{n} e^{-0.18i} = e^{-1.8k} \sum_{i=0}^{n-10k} e^{-0.18i} \leq e^{-1.8k} \sum_{i=0}^{\infty} e^{-0.18i} = e^{-1.8k} \frac{1}{1 - e^{-0.18}}$$

So we have that $\Pr[Y] \leq 7e^{-1.8k}$.

We have k processes, so the probability that in any execution α of algorithm ARTA for any process p and any state $s \in \alpha$, $|s.\text{DONE}_p| > 20k$ is less than $7ke^{-1.8k} \leq 7e^{-1.8k+\log k}$.

□

Now we need to combine the main lemmas from above in order to bound work complexity. More over we can bound space complexity with high probability using the lemma above. Clearly the DONE and FREE sets that each process keeps in each internal memory could be stored in the shared memory, if it is of interest to have O(1) local memory.

Theorem 8.14. *Algorithm* ARTA *has high probability work complexity* $O(n+k^2\log^3 k)$.

Proof. We have to count the work from calls to functions *progress*() and *defrag*(). We start with function *progress*().

As discussed above, the DONE and FREE sets are stored in some efficient tree data structure. So in function *progress*(), in lines 1 and 25, the calls to functions FREE.*randomElement*() and FREE.*remove*(*tail*,*head*) cost work $O(\log|\text{FREE}|)$ while the call to function DONE.*add*(*tail*,*head*) in line 26 costs work $O(\log|\text{DONE}|)$. From Lemma 8.13 with probability greater than $1-e^{-k}$ this work is $O(\log k)$. All the other lines of function *progress*() cost $O(1)$ work. The while loop in lines $7-14$ is executed once for each task performed by the function *progress*. So from all the calls to function *progress*() we have amortized work of $O(n)$ from the while loop in lines $7-14$. Moreover each call to function *progress*() results in additional $O(\log k)$ work.

From Lemma 8.12 we have that in any execution of algorithm ARTA there exist no more than $\frac{4(C+2)n}{\log k}+5(C+2)k^2\log^2 k$ calls to the function *progress* with probability greater than $1-2k^{-a}$, for $C>2(a+1)$. So with probability greater than $1-(2k^{-a}+e^{-k})$ the total work from calls to function progress is $O(n+k^2\log^3 k)$.

For each call to the function *progress*() from a process p there exists at most one call to function *defrag*(). In function *defrag*() we have calls to functions DONE.*randomInterval*() and DONE.*add*(*tail*,*head*) in lines 1 and 15, and to function FREE.*remove*(*tail*,*head*) in line 14. As discussed above these calls cost work $O(\log k)$. All other lines of function *defrag*() cost work $O(1)$. In each call of function *defrag*()

line 1 is executed only once. Line 14 and 15 are inside the while loop in lines $2-16$. This loop is executed once for each interval of completed tasks discovered by function *defrag*(). In *defrag*() we start from an interval of completed tasks that is known to process p and inserted to its DONE set, and look one position to the left of its tail, in order to discover a new interval of completed tasks. Since tasks are performed in a rightwards direction and we discover intervals looking to the left of the tail of an interval, any interval that we discover with *defrag*() cannot expand further, and thus is a completed interval. Any such interval can be discovered only once by a process p. As discussed in the proof of Lemma 8.12, from Lemmas 8.10 and 8.11 we have that the set of completed tasks is fragmented in less than $\left(\frac{4n}{k^2\log^2 k}+5\log k\right)(Ck\log k+k)=\frac{4(C+1)n}{k\log k}+5(C+1)k\log^2 k$ intervals, with probability greater than $1-2k^{-a}$. Since the while loop in function *defrag*() discovers intervals that cannot be expanded any more, and each execution of the loop costs $\mathrm{O}(\log k)$ work, the total work from all the executions of the loop by all processes is $\mathrm{O}(n+k^2\log^3 k)$. Moreover the work from the execution of line 1 of function *defrag*() is also $\mathrm{O}(n+k^2\log^3 k)$ since we have no more than $\frac{4(C+2)n}{\log k}+5(C+2)k^2\log^2 k$ calls to function *defrag*().

From the discussion above we get that algorithm ARTA has work complexity $\mathrm{O}(n+k^2\log^3 k)$, with probability greater than $1-(2k^{-a}+e^{-k})$. □

8.3.3 Space Complexity

We finally examine the space requirements of algorithm ARTA. From the discussion on the shared variables, we have that share memory is split in a vector W with n elements. Each elements has 1 bit that supports an atomic test-and-set operation and the *head* and *tail* pointers. Shared and internal memory cells are of size $\mathrm{O}(\log n)$. Moreover we assume the atomic test-and-set operation is provided by the hardware and is not implemented a

probabilistic object. Thus algorithm ARTA requires 3n shared memory space. In terms of internal memory requirements, from Lemma 8.13 we have that both set FREE and DONE for a process p in any state s have less than $20k$ elements with high probability. This gives us the next Theorem:

Theorem 8.15. *Algorithm* ARTA *requires* $O(n)$ *share memory space. Moreover each process uses* $O(k)$ *internal memory space, with high probability.*

Proof. As discussed above algorithm ARTA requires 3n shared memory space. Moreover from Lemma 8.13 we have that both set FREE and DONE for a process p in any state s have less than $20k$ elements with probability greater than $1-e^{-k}$. Since the sets are stored in a binary tree data structure the space required by a process p in order to keep sets FREE and DONE is $O(k)$ with probability greater than $1-e^{-k}$. □

9 Future Work

In this section we propose two specific directions for future work. The first one consists of work in the message passing model. The second one, deals with solutions for the at-most-once problem that focus on optimizing the memory usage of algorithms.

9.1 Message Passing Model

An important question arising in the context of the at-most-once problem, is how can the work presented in the asynchronous shared memory model, be applied in message passing systems. We propose the emulation of the shared-memory algorithms we designed, in asynchronous message passing systems. Working on the message passing model will be a continuation and extension of the work by Di Crescenzo and Kiayias [12], which was also the main motivation for this thesis.

Below we outline the methodology that can be used in order to emulate our shared-memory algorithms in the asynchronous message passing model. Moreover, for some of our algorithms, we conjecture the performance of such emulation. Using quorum systems we can emulate atomic shared memory robustly in message passing systems as shown by Attiya *et al.* [5]. In order to implement an atomic multi-reader/ multi-writer memory service, we can use a simplified version of the work presented by Lynch and Shvartsman [38]. Based on the above and using similar techniques with the ones deployed by Kowalski *et al.* [31], if the adversary is constrained to not disable the quorums used by the atomic memory service, we can solve the at-most-once problem by emulating our at-most-once algorithms, in a message passing system with bounded worst case message latency d and bounded worst case processor response time to messages e, where d, e are unknown to processors, maintaining the same performance (as the original aglorithms) in terms of effectiveness. We can then analyze the work and message complexity of our implementations in terms of d, e and K the size of the largest quorum configuration in the shared memory service. We next discuss what we can expect from the emulation of the various shared memory algorithms presented in this thesis.

- Algorithm $\text{AO}_{m,n}$ presented in Section 4 has the characteristic that it uses boolean shared memory registers that can only switch from 0 to 1. This means that the techniques applied in [31] in order to emulate the Write-All algorithms X [9] and AWT [4] can be also applied for algorithm $\text{AO}_{m,n}$. We conjecture that with similar analysis, emulating $\text{AO}_{m,n}$ will give us the same effectiveness as in shared memory, work of $\text{O}(\max\{K,d,e\}(n+m\log m))$ and message complexity of $\text{O}(K(n+m\log m))$. Moreover the correctness of the algorithm will be maintained even for an unrestricted adversary.

  It is interesting to observe here, that in the Write-All algorithms X [9] and AWT [4]

emulated in [31], as well as in the algorithm $\text{AO}_{m,n}$ that we intend to emulate, the multi-reader/ multi-writer memory cells change only once from 0 to 1, while subsequent writes only attempt to write 1 to the already set memory cell. Clearly for such write operations there is no need for two round writes. Moreover it might be worth exploring whether there exist semi-fast, or fast implementations for multi-reader/ multi-writer memory cells that support only a one-shot switch from 0 to 1, in the message passing model. If such constructions exist, then we can have a tighter analysis for the algorithms emulated in [31], as well as for the emulation of algorithm $\text{AO}_{m,n}$. Asymptotically the result will remain the same, but the constants would be reduced. A positive result on the existence of shared memory services with such attributes, has also value outside the scope of the proposed thesis.

- The emulation of KK_β presented in Section 5 can also be performed using similar techniques like the ones in [31]. It is interesting to note here that KK_β does not need an atomic multi-reader/ multi-writer memory service, but can instead use an atomic single-writer/ multi-reader memory service like the one presented by Georgiou *et al.* [16, 17]. Such a service provides fast writes and semi-fast reads, in contrast with the two round writes and reads required by a multi-reader/ multi-writer implementation.

  It is interesting to study whether the Write-All algorithm WA_IterativeKK(ε) presented in Section 6 can be also implemented using atomic single-writer/ multi-reader memory, thus extending the work in [31] by using a simpler memory service. It will be interesting to analyze the performance of such an emulation and get a tight bound on the number of two-round reads needed by the semi-fast implementation. It is also possible to perform a probabilistic analysis in order to bound the number of slow reads using similar techniques with the ones in [15].

- Randomized algorithm RA presented in Section 7 can also be emulated in the same way. We should also be able to emulate RA using an atomic single-writer/ multi-reader memory service. Algorithm RA limits multiple writer access to memory through the use of randomized test-and-set operations. The main challenge is making sure that the randomized test-and-set primitives can be implemented using single-writer/ multi-reader memory, which my also be of independent interest.

9.2 Space Complexity Bounds and Optimization

In the work presented in this dissertation, optimizing the use of shared memory was not explicitly considered. As a result the algorithms we presented above do not optimize in terms of space complexity. Moreover we are not aware of any work in the literature that focuses on optimizing space complexity for the (strong) at-most-once problem.

This is an important question, since in order to emulate shared memory algorithms in the message passing system using the methodology described in the previous subsection, for each memory cell, an atomic shared memory service needs to be implemented by a set of processes that act as a quorum. This demonstrates that reducing the space requirements of at-most-once algorithms for the shared memory model, is worth pursuing. We would like to explore what is the lower bound on space complexity for the strong at-most-once problem, or for algorithms that solve the at-most-once problem with optimal effectiveness, in terms of shared memory cells used by an algorithm. We conjecture a $k \log n$ space complexity lower bound for the strong at-most-once problem, where k the processes participating in an execution. Proving such a lower bound could be achieved by showing that in a fail-stop synchronous shared memory, one cannot device algorithms for the strong at-most-once problem, using less than $k \log n$ bits of shared memory. A different approach in proving a lower bound for the space complexity of the strong at-most-once problem,

would be to resort in the literature for consensus 2 objects and the space lower bounds related to them. Moreover, giving a randomized algorithm that matches the $k \log n$ space complexity bound would be an interesting direction that we will consider. Modifying algorithm RA presented above could be a good starting point. A main challenge is implementing randomized test-and-set operators with low space complexity (which may also be of independent interest), or avoiding using such operators, in favor of some other, more space efficient mechanism.

The question of space optimal solutions, under some strict performance guarantees may also be of interest in the context of the Write-All problem. We conjecture a similar lower bound on the space complexity of work optimal solutions for the Write-All problem, and conjecture that a trade-off exists between the space complexity and work complexity of Write-All solutions.

10 Summary of Open Problems

There exist various open problems left from this dissertation. We group them below by the type of questions they address:

- **Modeling**

  The asynchronous shared memory model is a high level abstraction that covers most multiprocessor and multi-core architectures. Still there are many architecture specific models that are general enough to have wide applicability and different enough to benefit from different algorithmic approaches. Architecture aware approaches can lead in more efficient designs in terms of effectiveness, work or space complexity. Exploring the (strong) at-most-once problem in *hierarchical shared memory* may lead in surprising new results. Of particular interest should be *Non-*

Uniform Memory Access multiprocessors. Moreover one can examine how *cache coherency* protocols could affect the design of algorithms. Different *consistency models* for cache coherency could benefit from different algorithmic strategies.

Another direction lies in examining simpler shared memory models. Examining the at-most-once problem in synchronous shared memory with fail-stop crashes, could provide useful intuition on the fundamental difficulties of the at-most-once problem.

A questions lies on how failure detection oracles can influence the design of solutions for the at-most-once problem. The upper bound on effectiveness for the at-most-once problem (Corollary 3.3) will still hold, but would it become easier to device tight solutions in terms of optimal effectiveness?

In terms of modeling, an obvious direction is towards message passing systems. There is a wide range of challenges in such an endeavor. New definitions are needed for the problem, as well as examination of what will be a meaningful message passing model for the problem. How fragmentation can affect solutions and whether it can be circumvented or not. Finally, since in order to simplify programming, message passing systems resort in using middle-ware architectures that implement asynchronous shared memory, it is interesting to explore asynchronous shared memory models that support weaker primitives than the atomic read/write registers we have used so far.

- **Related Problems**

  The natural generalization of the at-most-once problem is the at-most-k notion, where k is either a constant, or depends on m the number of processors. One then needs to examine how the impossibility results and the lower bounds for the at-

most-once problem transfer in the new setting. A strong notion of the at-most-k would also be worth examining. A different direction would be the definition of a weak at-most-once, or a weak at-most-k problem where the safety semantics can be violated to some controlled extend. Such a problem may have applicability in real systems, where jobs are at-most-once in nature, but one can afford some violation of the semantic.

It is also interesting to explore under what model a solution to a *do-all and at-most-k* problem can be achieved and how strong such a primitive is in terms of its consensus number. From the impossibility results on the at-most-once, such a problem is clearly not solvable if k failures are allowed. So a question arises on whether there exist models, where this impossibility can be lifted.

Finally dynamic versions of the at-most-once problem can be examined, where either the tasks are dispatched dynamically, or the processes arrive in a dynamic way, or both. Such a setting imposes different challenges than the static model and is closer to real applications.

- **Impossibility Results and Lower Bounds**

  There is a need for an impossibility result on wait-free deterministic solutions that use atomic read/write registers for the at-most-once problem, similar to the one for the strong at-most-once. We need to explore what constrains on the effectiveness of solutions could give us such an impossibility result. We conjecture that such a result exists if you require optimal effectiveness. Can we get something stronger, proving that even sub-optimal wait-free deterministic solutions cannot be achieved?

  We are currently also missing lower bounds on the work of solutions with effectiveness near $n - f$. Establishing first an impossibility result on wait-free deterministic

solutions may help in asking the correct questions. For example, for what effectiveness should one seek such a lower bound?

Finally as discussed above lower bounds on the space complexity of algorithms with effectiveness near $n - f$ need to be established.

- **Collision Detection Algorithms**

  The multiprocess algorithm $\text{AO}_{m,n}$ presented in Section 4 scales well when it comes to work complexity, but does not scale well in terms of effectiveness when the number of processes increases. The algorithm applies two techniques. The first one is the collision avoidance strategy, and the second one is having jobs that can be performed by only specific groups of processes. This jobs provide starting points for processes when they join the execution of the algorithm. This means that algorithm $\text{AO}_{m,n}$ cannot be a solution for the strong at-most-once problem. Could we modify the algorithm, to make it a candidate solution for the strong at-most-once problem? One approach would be to have some join service in the algorithm, that provides starting points, for processes joining the execution.

  It would also be interesting to devise algorithms that focus more on the second technique applied by algorithm $\text{AO}_{m,n}$. We believe that such algorithms could have different characteristics than algorithm $\text{AO}_{m,n}$. Specifically we believe that they may scale better in terms of effectiveness when the number of processes increases.

- **Algorithm** KK_β

  There is an open question concerning the work complexity of algorithm KK_β presented in Section 5 for $\beta < 3m^2$. We expect that such an analysis will be quite involved. A positive result for the work complexity of KK_β for $\beta < 3m^2$, could lead in interesting new iterative solutions for both the at-most-once problem and

the write-all problem. A different direction would be to perform a probabilistic work complexity analysis for $\beta < 3m^2$ using similar techniques as the ones applied in [15].

There still exists an effectiveness gap between the shown effectiveness of $n-2m+2$ of algorithm KK_β and the known effectiveness bound of $n-m+1$. It would be interesting to study if this can be bridged for wait-free deterministic algorithms. If not, it would be interesting to device deterministic algorithms that solve the problem when some progress requirements are met.

Algorithm KK_β has a $\mathrm{O}(nm\log n)$ space complexity for shared memory. By using a linked list instead of a matrix in which processes announce the jobs they complete, algorithm KK_β can be modified to have $\mathrm{O}(n\log n)$ shared memory space complexity. The correctness analysis of the modified algorithm will be quite involved, since now the cells of the linked list can be potentially written by any process, and the correctness of the implementation relies in the at-most-once property of job execution.

Algorithm KK_β can be readily modified, in order to provide a solution for a dynamic setting or streaming setting, where jobs are not known a priory and may arrive indefinitely. We conjecture that if infinite jobs arrive, then infinite jobs will be performed, and that a process failure may block exactly 1 job for infinite steps, while all other jobs will be eventually executed provided that more jobs arrive. It would be interesting to reduce the space requirements in such a setting, in order to make the solution practical.

A different direction would be to examine whether algorithm KK_β can be modified in order to provide a k-adaptive deterministic solution for the at-most-once problem.

Finally, it is possible to fine grain algorithm IterativeKK(ε) presented in Section 6, in order to get a tighter effectiveness of $n-(3m^2+m-2)$ without impacting work complexity considerably. Proving correctness for such a solution will be quite involved.

- **Strong At-Most-Once Solutions**

  One question that arises from Theorem 3.7, is what kind of deterministic solutions can we expect for the strong at-most-once problem. Clearly such solutions cannot be wait free. It is interesting to study what progress requirements are reasonable for such solutions. Partial synchrony or failure detectors could be possible ways to circumvent the impossibility result.

  It would be interesting to study a lower bound on work for randomized algorithms, as well as a lower bound on work for k-adaptive algorithms both in the context of the at-most-once problem, as well as the Write-All. The lower bound of $\Omega(n+p\log n)$ from [43] is a good indication, but how does this change for k-adaptive algorithms? Can solutions of $\Omega(n+k\log n)$ be achieved?

- **Applications**

  One different direction would be to examine how real applications can be impacted by the results of the current thesis. As mentioned above, in message passing applications, in order to simplify programming, shared memory middle-ware architectures are used. Such applications could be in the context of wireless sensor and actuator networks, as part of pervasive computing, energy efficient buildings, smart home, industrial automation and robotics. Software solutions for at-most-once semantics can allow for simpler and cheaper actuators, while allowing for multiple actuators to control the same tasks, increasing the reliability and flexibility of the deployed systems.

References

[1] Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects. In *PODC*, pages 159–170. ACM, 1993.

[2] D. Alistarh, M.A. Bender, S. Gilbert, and R. Guerraoui. How to allocate tasks asynchronously. In *Foundations of Computer Science (FOCS), 2012 IEEE 53rd Annual Symposium on*, pages 331 –340, Oct. 2012.

[3] Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In *DISC*, pages 94–108, 2010.

[4] Richard J. Anderson and H. Woll. Algorithms for the certified write-all problem. *SIAM J. Computing*, 26(5):1277–1283, 1997.

[5] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, January 1995.

[6] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, 1990.

[7] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[8] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[9] Jonathan F. Buss, Paris C. Kanellakis, Prabhakar Ragde, and Alexander A. Shvartsman. Parallel algorithms with processor failures and delays. *Journal of Algorithms*, 20(1):45–86, 1996.

[10] Soma Chaudhuri, Brian A. Coan, and Jennifer L. Welch. Using adaptive timeouts to achieve at-most-once message delivery. *Distrib. Comput.*, 9(3):109–117, 1995.

[11] Bogdan S. Chlebus and Dariusz R. Kowalski. Cooperative asynchronous update of shared memory. In *STOC*, pages 733–739, 2005.

[12] Giovanni Di Crescenzo and Aggelos Kiayias. Asynchronous perfectly secure communication over one-time pads. In *Proc. of 32nd International Colloquium on Automata, Languages and Programming(ICALP '05)*, pages 216–227. Springer, 2005.

[13] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[14] Matthias Fitzi, Jesper Bus Nielsen, and Stefan Wolf. How to share a key. In *Allerton Conference on Communication, Control, and Computing 2007*, 2007.

[15] Chryssis Georgiou, Sotiris Kentros, Nicolas C. Nicolaou, and Alexander A. Shvartsmanand. Analyzing the number of slow reads for semifast atomic read/write register implementations. In *Proc. of the 21st International Conference on Parallel and Ditributed Computing and Systems(PDCS 2009)*, pages 229–236, 2009.

[16] Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. On the robustness of (semi) fast quorum-based implementations of atomic shared memory. In *PODC*, page 425, 2008.

[17] Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. *J. Parallel Distrib. Comput.*, 69(1):62–79, 2009.

[18] Chryssis Georgiou and Alexander A. Shvartsman. *Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity*. Springer, 2008.

[19] Chryssis Georgiou and Alexander A. Shvartsman. *Cooperative Task-Oriented Computing: Algorithms and Complexity*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2011.

[20] Kenneth J. Goldman and Nancy A. Lynch. Modelling shared state in a shared action model. In *Logic in Computer Science*, pages 450–463, 1990.

[21] Jan Groote, Wim Hesselink, Sjouke Mauw, and Rogier Vermeulen. An algorithm for the asynchronous write-all problem based on process collision. *Distributed Computing*, 14(2):75–81, 2001.

[22] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science(FOCS)*, pages 8–21, 1978.

[23] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13:124–149, 1991.

[24] Keren Censor Hillel. Multi-sided shared coins and randomized set-agreement. In *Proc. of the 22nd ACM Symp. on Parallel Algorithms and Architectures (SPAA'10)*, pages 60–68, 2010.

[25] Paris Christos Kanellakis and Alex Allister Shvartsman. *Fault-Tolerant Parallel Computaion*. Kluwer Academic Publishers, 1997.

[26] Sotiris Kentros, Chadi Kari, and Aggelos Kiayias. The strong at-most-once problem. In *DISC*, pages 390–404, 2012.

[27] Sotiris Kentros and Aggelos Kiayias. Solving the at-most-once problem with nearly optimal effectiveness. In *ICDCN*, pages 122–137, 2012.

[28] Sotiris Kentros and Aggelos Kiayias. Solving the at-most-once problem with nearly optimal effectiveness. *Theor. Comput. Sci.*, 496:69–88, 2013.

[29] Sotiris Kentros, Aggelos Kiayias, Nicolas C. Nicolaou, and Alexander A. Shvartsman. At-most-once semantics in asynchronous shared memory. In *DISC*, pages 258–273, 2009.

[30] Dariusz Kowalski, Peter M. Musial, and Alexander A. Shvartsman. Explicit combinatorial structures for cooperative distributed algorithms. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, ICDCS '05, pages 49–58, Washington, DC, USA, 2005. IEEE Computer Society.

[31] Dariusz R. Kowalski, Mariam Momenzadeh, and Alexander A. Shvartsman. Emulating shared-memory do-all algorithms in asynchronous message-passing systems. *J. Parallel Distrib. Comput.*, 70(6):699–705, June 2010.

[32] Dariusz R. Kowalski and Alexander A. Shvartsman. Writing-all deterministically and optimally using a nontrivial number of asynchronous processors. *ACM Transactions on Algorithms*, 4(3), 2008.

[33] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[34] Butler W. Lampson, Nancy A. Lynch, and Jrgen F. S-Andersen. Correctness of at-most-once message delivery protocols. In *Proc. of the IFIP TC6/WG6.1 6th International Conference on Formal Description Techniques(FORTE '93)*, pages 385–400. North-Holland Publishing Co., 1994.

[35] Kwei-Jay Lin and John D. Gannon. Atomic remote procedure call. *IEEE Trans. Softw. Eng.*, 11(10):1126–1135, 1985.

[36] Barbara Liskov. Distributed programming in argus. *Commun. ACM*, 31(3):300–312, 1988.

[37] Barbara Liskov, Liuba Shrira, and John Wroclawski. Efficient at-most-once messages based on synchronized clocks. *ACM Trans. Comput. Syst.*, 9(2):125–142, 1991.

[38] N. Lynch and A.A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proc. of 16th International Symp. on Distributed Computing(DISC'02)*, pages 173–190, 2002.

[39] Nancy Lynch and Mark Tuttle. An introduction to input/output automata. *CWI-Quarterly*, pages 219–246, 1989.

[40] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[41] Grzegorz Malewicz. A work-optimal deterministic algorithm for the certified write-all problem with a nontrivial number of asynchronous processors. *SIAM J. Comput.*, 34(4):993–1024, 2005.

[42] Charles Martel and Ramesh Subramonian. On the complexity of certified write-all algorithms. *J. Algorithms*, 16:361–387, May 1994.

[43] Charles Martel and Ramesh Subramonian. On the complexity of certified write-all algorithms. *J. Algorithms*, 16(3):361–387, May 1994.

[44] F. Panzieri and S.K. Shrivastava. Rajdoot: A remote procedure call mechanism supporting orphan detection and killing. *IEEE Transactions on Software Engineering*, 14(1):30–37, 1988.

[45] Alfred Z. Spector. Performing remote operations efficiently on a local computer network. *Commun. ACM*, 25(4):246–260, 1982.

[46] R. W. Watson. The delta-t transport protocol: Features and experience. In *Proc. of the 14th Conf. on Local Computer Networks*, pages 399–407, 1989.